A demand driven multiprocessor.

Bakti, Zulkifli Abdul Kadir

Award date:
1985

Awarding institution:
University of Bath

A DEMAND DRIVEN MULTIPROCESSOR

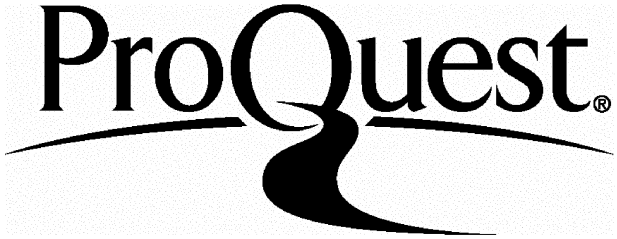
submitted by

Zulkifli Abdul Kadir Bakti

for the degree of Ph.D.

of the University of Bath

1985



ACKNOWLEDGEMENTS

The author wishes to express his gratitude to Dr. P.J. Willis and Professor J.P. Fitch for their supervision of this work. He would also like to thank the other members of the computing group in the School of Mathematics, especially to Dr. D. Milford.

Appreciation is extended to the Science and Engineering Research Council for making a grant available under the Distributed Computing Systems (DCS) programme for financing the construction of the hardware.

Finally, his sincere thanks to Universiti Sains Malaysia, Penang for providing the financial support.

SUMMARY

It is thought that fast low cost computers can be built by employing large numbers of cheap microprocessors working together in a system. However increasing the number of microprocessors in a parallel computer system may not produce a linear increase in performance for general purpose programming. The problems seem to lie in the communication between processors and the method of exploiting parallelism.

A multiprocessor system was constructed using six MC68000 microprocessors. The problems of communication and exploiting parallelism were tackled in the design of the multiprocessor system.

The component processors in a multiprocessor system communicate with each other through a communication channel. It is essential that the communication hardware has a high bandwidth. A fast communication hardware was implemented based on a two port shared memory.

One method of extracting parallelism in a computing problem is by using divide and conquer. A software system was developed that enables the multiprocessor to exploit parallelism derived by the divide and conquer method. A software kernel is employed to manage the scheduling of parallel tasks to processors and the communication between processors. The mode of computation is based on the demand driven model.

CONTENTS

CHAPTER 1 INTRODUCTION

1.1 Introduction

Computers are used as tools in some applications and as components of systems in other applications. In some critical applications high speed is of considerable importance. It is thought that employing more than one processor working in parallel to each other in a computer system will increase the processing speed.

The component processors in a parallel computer system can be off the shelf microprocessors (8) or specially made. The simplest form of parallel computer system consists of a collection of microcomputers linked to each other by means of a communication path. The communication path could either be a shared memory or parallel or serial data link. Shared memory architectures are classified as tightly coupled and data link architectures as loosely coupled. In a multiprocessor system the individual processors are themselves complete computer systems, possibly having an ample amount of memory and some input/output capabilities. The notion of parallel processing is to have several processors cooperating towards the solution of a common problem. The vehicle through which the processors are able to cooperate with one another is provided by the interprocessor communication. Typical of this type of parallel computer or multiprocessor system is that there is no centralised control.

At the other end of the scale, a highly synchronous form of parallel computer replicates only the arithmetic unit but not the control unit.AHigh degree of parallelism is achieved when the same operation is to be performed on a multitude of data elements. However if the number of processing units available is less than the number of the data elements part of the processing has to be done serially. The advantage of synchronous control is simpler communication as there is little overhead involved in setting up.

With the above examples of parallel computers, one thing that they share is the Von Neumann model with regard to their programmability. Some circles within the computing community believe that this restricts the efficiency of parallel computers. New computer architectures based on data flow, functional and reduction models should be capable of exploiting parallel processing to the fullest.

Generally the approaches to research in parallel architectures are to concentrate on off the shelf technology because of low cost or to use specialised hardware in order fully to characterise the computing models. The line of approach adopted in this thesis is to find new methods of exploiting parallelism in a multiprocessor system utilising an ensemble of microprocessors. The factors that are of interest are the method of interprocessor interface and the software organisation. Both factors will determine the

performance of the system. With interprocessor communication the topology of processor to processor communication based on a system of graphs needs investigation. With software organisation, a suitable computing model and also the method of exploiting parallelism needs choosing.

1.2 Approaches to parallelism

The sequential execution of one instruction at a time and the updating of a state dependent data structure is the basis for the uniprocessor. The imperative programming languages such as Fortran, Pascal etc. closely emulate this scheme. This will make these languages unsuitable for programming multiprocessors. Functional languages such as pure Lisp, do not depend on state dependent variables and there is the possibility for expressing parallel evaluation implicitly. However the limitations of imperative languages do not exclude them from being the basis for some concurrent languages and the cleanness of functional languages does not make them into a universal multiprocessor programming language. There are reasons for this being the case. The use of microprocessors as component processors in the multiprocessor system is the main reason. Another reason is the nature of the applications for these

systems.

In real time applications, a suitable model can be built by mappping the processes into individual processors. It is sufficient to consider that all processors are running their own programs or task. To supervise an orderly interaction between tasks, new constructs are introduced. This is the basis for some concurrent languages such as Concurrent Pascal (45), Ada (54) and Occam (25). The advantage of this approach is that very little overhead is needed apart from that required for establishing communication. The use of functional languages will extend the capability of multiprocessors to general purpose applications. The disadvantage is the hardware based on off the shelf microprocessor is not capable of emulating the functional model directly. An extra level of software is required to interpret the functional model.

1.2.1 Multimicroprocessor system

Programming multiprocessors based on interacting sequential processes is the first approach mentioned in the last section. Originally this method was developed for operating system and real time system programming. Separating the various functions in an operating system into individual processes that coexist in time results in a more efficient and easily maintainable program. The virtual parallel processes are simulated by time multiplexing the physical processor.

The availability of cheap microprocessors tempts many system designers to swap the virtual processors for real processors. For existing applications this transition provides an acceptable gain in speed performance although with more hardware complexity.

1.2.2 Concurrent language concept

The main issues which a concurrent language highlights are the concept of task and communication. The communication can occur through a common area shared by the tasks or by the passing of messages through a channel from one task to another. For communication to be effective there are certain conditions that have to be imposed.

An early concurrent language such as Concurrent Pascal provides shared memory oriented communication. This is due to the multitasking type of applications on uniprocessors. The techniques developed for this application can be applied wholesale to a shared memory architecture.

With shared memory type of communications, the most important aspect that needs to be tackled is guaranteeing a determinate access to the shared structures. Two methods which have gained wide acceptance are based on semaphore (44) and monitors (45).

1.2.3 Semaphore

A semaphore S is an integer variable which is common to all the processes involved. Initially S is assigned some value. Associated with this semaphore is a queue which holds the names of the processes. Two operations only are allowed on the semaphore. They are Wait(S) and Signal(S), abbreviated to P(S) and V(S) respectively by Dijkstra. The value S, is decremented when a process executes a Wait(S). If the value of S is negative the process is blocked from execution. This is done by putting the process on the queue. If the value of S is non-negative, the process is allowed to continue without delay. A process executes Signal(S). when it wants to release control of the shared data structure. The operation Signal(S) increments S. If the value of S is negative, the process at the head of the queue is scheduled for execution. A simple case is a binary semaphore, which can only assume binary values and deal with two processes. A queue is not needed as the delay operations can be performed by a simple busy loop.

1.2.4 Monitor

The monitor provides a higher level of abstraction than the semaphore. This allows operations on a shared data to be more structured. The monitor defines a set of shared data structures and a collection of procedures that can perform operations on this data.

The procedures inside the monitor are accessible to all the processes. The processes are not allowed to operate on the shared variables directly but can only do so through the monitor procedures. The executions of monitor procedures are mutually exclusive. The process that is executing a monitor procedure has exclusive control. Other processes can only access the procedure after the first process has released control. A process can give up a monitor procedure in two ways. First, by terminating the execution of the procedure. The second method is by performing two complementary operations on a conditional variable, Cond. The operations are Delay(Cond) and Continue(Cond). When a Delay(Cond) inside a monitor procedure is executed by another process, a process that is currently held on a queue is retrieved and its execution resumed.

1.2.5 Message based communication

In message based communication a channel is defined over which communication can takes place. A writer sends information through a channel. A reader receives information from a channel. Provided both the writer and reader specify the same channel, communication between the two is said to be established. Again synchronisation is necessary between the writer and reader. A concept called rendezvous is used in message based communication for this purpose. Transfer of information over the channel can only occur when both

the writer and the reader meet inside the channel. If either of them arrives earlier than the other, it must wait for the other to arrive. In extended rendezvous both the writer and reader maintain synchronisation for an extended period before departing. Communication Sequential Process(CSP) (24) describes a formalism for message based communication.

Incorporating the notion of message based communication in a high level language is easy. This can be done using two system procedures -

SEND(ch,data) and RECEIVE(ch,data)

where ch is the channel number and data the value send or received.

Ada and Occam are two languages that are based on message passing.

Physically a hardware link between two processors can represent the channel. Therefore Ada and Occam are the ideal languages to program loosely coupled multiprocessors. However shared memory multiprocessors can also use message passing communication.

1.2.6 Designing parallel programs

The parallel programming constructs described in the previous sections do not determine the method of designing parallel programs. Structured programming techniques have been developed for sequential languages. A similar technique should be applicable to concurrent programs. In structured programming the

program is decompose into smaller procedures. In a sequential program there is only a single thread of control. In concurrent programming the notion of decomposing the program into procedures or processes is still valid but the thread of control is capable of replicating (17)(19). The processes that reside on parallel threads are capable of being executed on parallel processors. The parallel processes may not be totally independent of each other. This is where interprocessor communication has to take place.

1.3 Array and vector processors

Array and vector processors are based on synchronous architectures. Using a typical sequential language like Fortran, there are certain aspects of parallelism that can be exploited. However this potential is limited to some areas of application such as numerical computing. In numerical computation, a lot of array and matrix manipulation is involved. If the elements of an array or matrix are independent of each other, simultaneous operations on all the data elements are possible. In normal sequential programming loops are used to work on the data elements. Parallel evaluation of the data elements has the effect of unfolding the loops.

Programs written in ordinary Fortran are automatically translated by compilers to produce code for the parallel machine. To make the translation task

easier, special Fortran style languages are used. Examples of such languages are CFT (48) for the Cray-1 and Ivtran (43) for the Illiac IV. Special constructs are introduced, for example the parallel assignment of data elements in Actus (42).

Array and vector architectures to date are the most successful parallel architectures. There are a number of commercial machines in this class, the so called supercomputers. Two examples of array and vector computers were mentioned in the previous paragraph, the Cray-1 and the Illiac IV (46). Further examples are CDC Star-100 (47) and ICL DAP (49).

1.4 Non Von Neumann architectures

It was found that parallelism cannot be exploited efficiently on conventional architecture (6). Various machine architectures and computing models have been proposed as alternatives to the conventional Von Neumann machine. The intrinsic characteristic of these architectures is that they should be capable of expressing parallelism naturally.

1.4.1 Data flow architectures

A data flow program is a system of graphs where the nodes represent evaluation and the edge or arcs represent the carriers for the arguments. A simple data

flow graph is shown in fig(1.1). The node or actor has two inputs and one output. An actor is said to be fireable if both inputs are valid and the output is empty. The relationship between the inputs and the output is dependent on the function of the actor. Actors that can only do binary operations are not sufficient to realise a complete computing model. Unary operations are supported by actors having one input and one output. To support conditionals, two type of actors are required, switch and merge. A switch actor has two inputs and two outputs. One of the inputs accepts predicate values. A data from the other input is directed to either output depending on the condition of the predicate input. A merge actor has three inputs and one output. One of the inputs receives the same predicate as the switch actor. The data token at either of the remaining inputs is directed to the output depending on the predicate input.

On the language side, there are special languages developed which would compile directly into a system of data flow graphs. Two such languages are VAL (38)(39) and SISAL (40). They are different from the normal imperative languages. Basically only single assignment is allowed.

The data flow has created a possibility of a computer without a program counter. The actors can be realised directly in hardware which could replace the basic logic building blocks. What will result is a

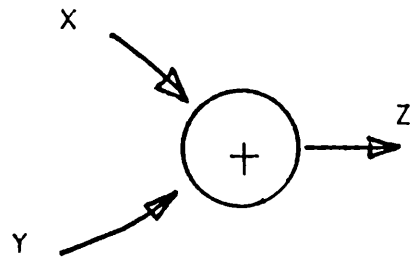


FIG. 1.1 A DATA FLOW ACTOR

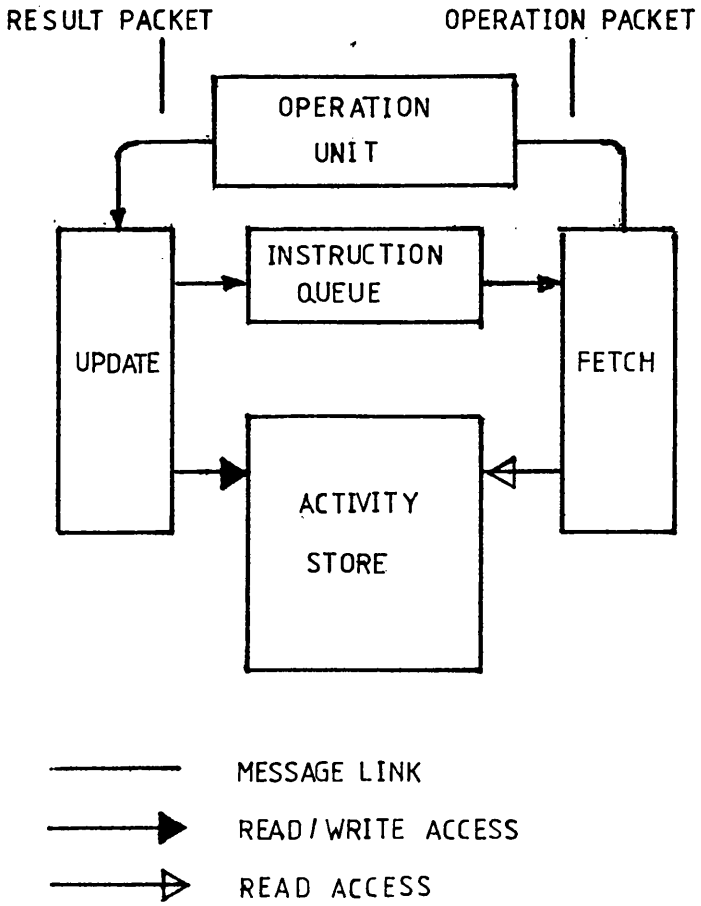


FIG. 1.2 BASIC DATA FLOW MACHINE

piece of computer program totally realised by an interconnection of hardware. Although this will restrict programming in the normal sense, it can offer true high speed performance for some applications. Albeit this has not gained wide acceptability apart from a device manufactured by NEC Electronic Incorporation (50). The device designated uPD7281 is a VLSI implementation of the data flow logics.

The present research trend is to develop a data flow architecture as a general purpose machine. The approach is to emulate the data flow machine using a high speed bit slice microprocessor. The actors are represented in memory as activity templates. The activity templates are grouped together in the activity store. A unique address is required for referencing a template.

A basic execution mechanism for a data flow processor due to Dennis (16)(27) is shown in fig (1.2). The data flow program is held as a system of activity templates in the activity store. The instruction queue contains the addresses of fireable activity templates. The fetch function retrieves an instruction from the queue. The instruction will specify an address of an activity template. This activity template is fetched from the activity store and made into an operation packet. The operation unit will execute the instruction to produce a result packet. The update unit will pass the result to the destination templates. If this result

causes the destination to be fireable, the address of the destination template will be placed on the queue.

A data flow multiprocessor consists of a collection of data flow processing elements. The combination of all the activity store will be asigned to a single address space. A communication network is used to transmit results to non-local activity templates. This network also works as a router by routing the packets to their appropriate destinations.

The Manchester Dataflow (20) follows the same principles of the dataflow machine of Dennis. However the Manchester dataflow introduces token labelling as a means of supporting reentrant code structures.

1.4.2 Reduction computers

Parallelism is available in a functional language at no extra cost. Consider a function which has several arguments and the arguments themselves are function calls. Before the values for the arguments can be used, the arguments have to be evaluated. If there are more than one argument, argument evaluations can be done in parallel. The process of reducing the arguments to useable values is called reduction. Reduction can either be string or graph reduction. In string reduction the process is done by redrawing a new instruction stream for each reduction. In graph reduction, the graph representing the computation is modified for each reduction.

The SKIM reduction machine (13) at Cambridge, uses a combinary logic or combinators to represent programs. Turner (10) originally developed the scheme of using combinators in applicative programming. The idea of combinators is to remove bound variables in applicative programs. The internal representation of the program is by cells of two elements.

A combinator system can be built typically using five symbols S,K,I,B,C which represent functions with reduction rules satisfying -

K xy = x

S fgx = fx (gx)

I x = x

B fgx = f (gx)

C fgx = fxg

Fig 1.3a shows a graph representing an expression (x+1)*(x-1) where x=7. The textual form of the same graph is S(B times(C plus 1))(C minus 1)7. Figures 1.3a to 1.3g show the steps in reducing the program graph for the above expression. It can be seen that after the S reduction there is a branching in the graph. There is a possibility of parallel reduction from this point onwards. What the graph represents here is that the two sub-expressions, (x+1) and (x-1) can be evaluated in parallel. There still is a problem in recognising when this can be done safely. For this reason many of these systems, including SKIM, ignore parallelism for the security of normal order reduction.

GRAPH BEFORE REDUCTION

(S(B×(C+1)(C-1)7)

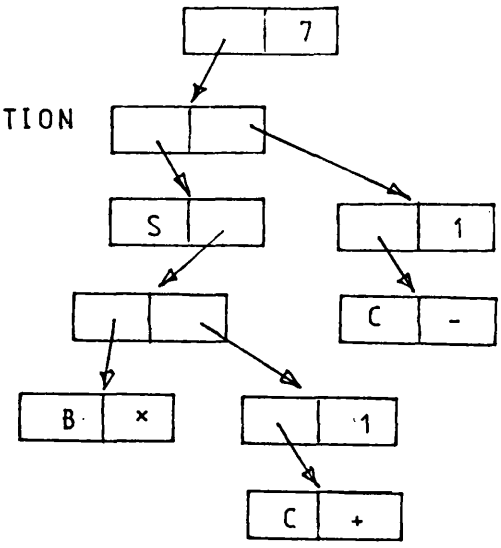


FIG. 1.3a (S(B×(C+1)(C-1)7)

APPLY S

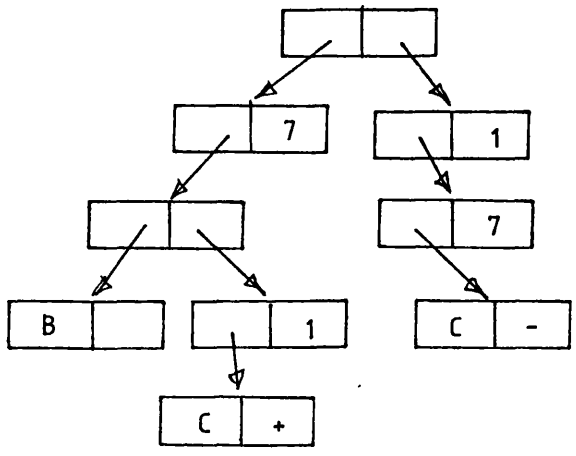


FIG. 1.3b (B×(C+1)7)((C-1)7)

APPLY B

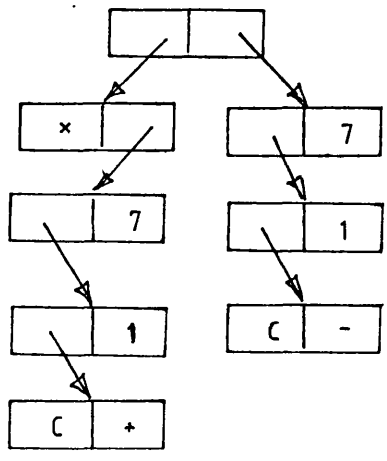


FIG. 1.3c (×(C+1)7)((C-1)7)

APPLY C

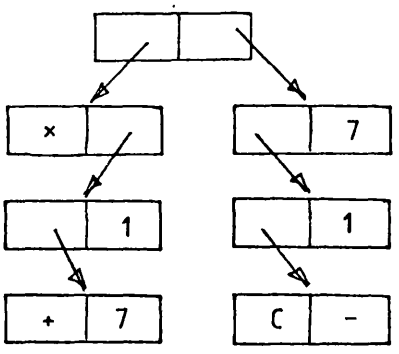


FIG. 1.3d (×(+ 7 1))((C-1) 7)

APPLY +

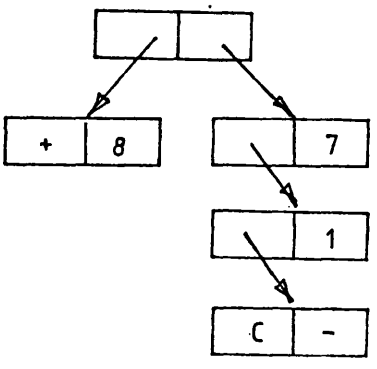


FIG. 1.3e (× 8 ((C-1) 7)

APPLY C

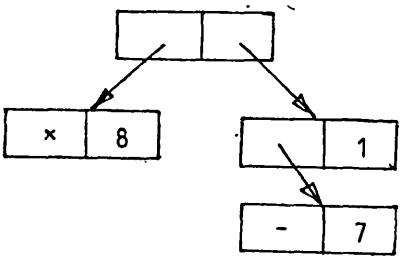


FIG. 1.3f (× 8 (- 7 1))

APPLY ×

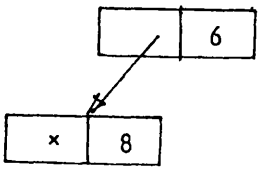


FIG. 1.3g (× 8 6)

The ALICE machine of Imperial College (1) also uses graph reduction. However it differs from the SKIM machine in that it does not use combinators. Furthermore the graph is represented by a system of packets. A packet consists of three primary and secondary fields. The primary fields are identifier, function and arguments list. The secondary fields are status, reference count and signal list. The execution node information is contained in the primary fields. The identifier signifies a unique address for the packet. The function field denotes the task of the node. The arguments list contains the references and values which form the input for the task. The secondary fields hold the necessary information for the control mechanism essential for execution. The status denotes the state of the task which can be active or suspended. The reference count is used in the garbage collection process. The signal list contains the information on the destination for the result of the task.

The ALICE machine exemplifies a typical demand driven computation. A need for a computation causes a demand packet to be created. This generates the node packet mentioned in the previous paragraph. The result is returned by the control packet.

In string reduction, programs are represented by a system of nested delimited strings (9)(23). The string is made up of characters from two alphabets. The first alphabet defines the character set for the delimiters.

The second alphabet defines the character set for the data.

A simple program fragment is -

(+ a1 a2) ...(s1)

The '+' determines the operations and, a1 and a2 can be nested substrings. The program fragment will then be -

(op1 (op2 b1 b2) (op3 c1 c2)) (s2)

The idea of string reduction is to take a string of the form (s2) and replace it with a string of the form (s1) but with a1 and a2 having simple values. Every time a reduction process is performed, a string is produced based on the original string. This is unlike graph reduction where the graph is redrawn by modifying the original structure.

A string reduction machine mode of operation is as follows. A string of the form (s2) is received at a processor. The string is scanned from left to right. Since there are nested strings present, the operation on the original string is delayed. Instead two more strings are produced which correspond to the nested substrings of the original. The evaluation of the smaller strings produces results that replace the nested substring with real values. The original string can then be evaluated and a final result produced.

1.6 Generating parallelism

In the multi microprocessor system of section 1.2.1, there is no debate whether the algorithm used

generates enough parallelism or not. In the context of real time application, the network of multiprocessors attempts to model the problem. In the vector and array processors, the unlooping of iteration generates the parallelism. In the non Von Neumann machine, although the expression of parallelism is natural the problem to be solved may not offer any parallelism. Unless the architecture is modelled directly by hardware, the availability of parallelism by multiple evaluation of argument does not justify the cost of communication and setting up the parallel processes. What is required is a system that can generate an exponential growth of parallelism. The situation is that parallelism can only be extracted and exploited from a problem that has the parallelism potential. This statement in a way reduces the applicability of using parallelism to a restricted set of problems only. Apart from numerical applications, the other area where speed is needed is in artificial intelligence systems. Array and vector processors are not suitable in this application because the nature of the problem does not involve numerical computation to a great extent. It is more suitable to use the reduction architecture for this type of application.

The divide and conquer method (14) solves a problem by continually subdividing it until the subproblem is small enough for direct evaluation. The subdivision produces a process tree. The results produced from the

leaf nodes are combined to form a partial result (if the parent node is an intermediate node) or the final result (if the parent node is a root node). A root node or intermediate node that is waiting for results from its subnodes is said to be in a suspended state.

Normally the type of the subproblem is the same as the original problem. The divide and conquer algorithm can be expressed more naturally as a recursive function or procedure. A control expression for the divide and conquer can be defined.

```
Program Divide-and-Conquer
 Const n =integer;
 Var A : array[1..n] of integer ;
     S : integer ;
 Function DandC (v,w : integer) : integer ;
  Var m,p,q ;
  Begin
   If Small(p,q) then DandC := G(p,q)
   Else Begin
          m := Divide(p,q) ;
          DandC := Combine(DandC(p,m),DandC(m+1,q))
        End
  End ;
 Begin
    S := DandC(1,n) ;
    .
 End.
```

Small is a boolean function which returns true if the problem cannot be subdivided further and returns false otherwise. Divide is a function which divides the problem. The Combine function combines the results produced by the subproblems.

The divide and conquer can be applied to a number of problems. The popular problem is sorting. NP complete problems (37) potentially can produce an enormous process tree. This makes NP complete problems solvable using divide and conquer. The Fast Fourier Transform algorithm (35) is an example of this class.

1.6 Interprocessor connection

Interprocessor communication is a very important aspect of multiprocessor implementation. In a system that employs hundreds or even thousands of microprocessors, the potential increase in speed can easily be upset by inefficient communication. With shared memory or shared bus architectures the performance begins to deterioriate with more than a few processors. This is due to the limited bandwidth of memory and bus that can be offered with present technology.

The present approach to interprocessor communication is to employ high speed data links between processors (53). There is a dedicated hardware interface for each link. The data link can be serial or parallel. The choice between the two is governed by the

physical constraints and whether the reduced transfer rate of a serial link is acceptable. Direct memory access control can be used for a very high speed communication but with added hardware complexity.

In principal a processor can communicate to any other processor in the system irrespective of whether the communication must be done via an intermediate processor, but this kind of communication must be restricted. The use of intermediate processors will tie up valuable processing resourses and will be very costly.

Having a fully interconnected path between all the processors is feasible for a system with relatively few number of processors. In a large system an interconnection strategy must be found that would be economical on the use of processors. One aspect that needs to be avoided is to resort to the use of intermediate processors for most communication.

Generally the nature of the problem determines the way parallelism can be exploited. To make the most efficient use of the parallelism, the hardware must be able readily to exploit it. For example the array and vector computers efficiently utilise the parallelism by being capable of modelling the execution. There are various topologies proposed, each having its own merits and suitability to the nature of problems that they can solve.

A binary tree of processors is capable of modelling

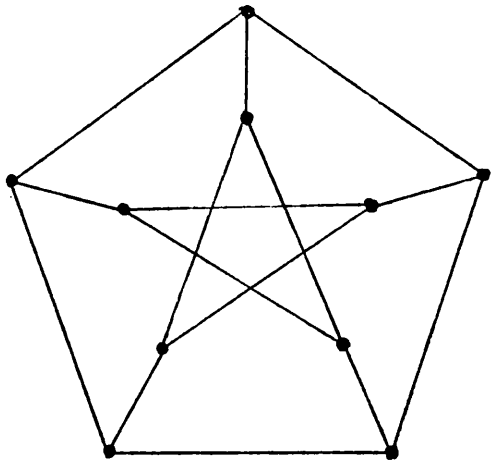


FIG. 1.4 PETERSEN'S GRAPH

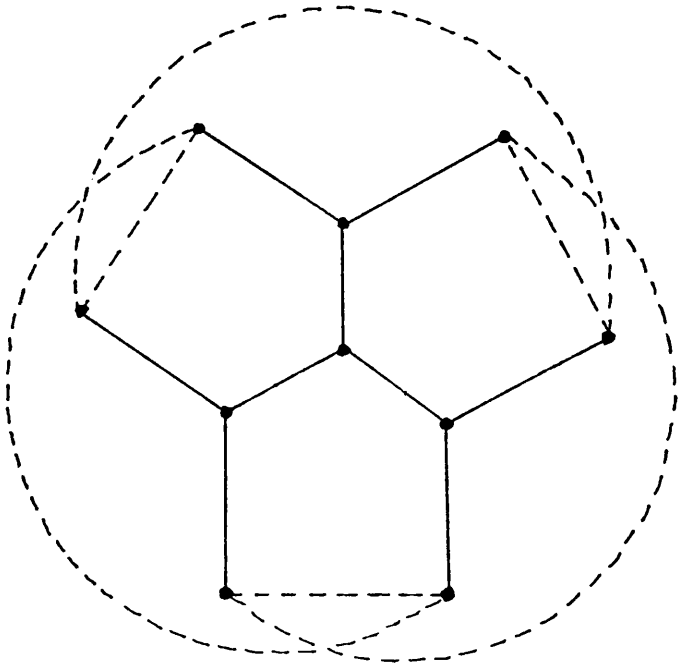


FIG. 1.5 PETERSEN'S GRAPH AS A TREE

the process trees of divide and conquer evaluation (55). The disadvantage is that a vast amount of processors are required once the depth of the tree exceeds a certain level. Processors will be idle during the suspended state of a node. This will amount to a very high wastage of valuable processing capabilities. Redeploying the system to another problem where subdivision is n ways is difficult as it entails hardware modifications.

An interprocessor interconnection scheme is proposed by Bowyer et al (7) based on a system of graphs. The graph chosen for the purpose is the Petersen's graph (fig. 1.4) which has a valency of 3 and girth 5. The symmetry of the Petersen's graph can be shown clearly in the redrawn diagram (fig. 1.5). The central node, chosen arbitrarily, can be seen to be the root node of three binary trees. The interesting properties of the Petersen graph is it has the maximum depth achievable for a graph having ten nodes. Assuming that a node in the graph represents a processor, a divide and conquer division will be done the most number of times before reaching the original root processor. The root processor would be idle and with the help of some software the root processor can be redeployed.

A trivalent graph of maximal girth can be employed for systems of different number of nodes. The criterion of maximum depth should still be upheld. For practical

reasons graphs with very high valency may not be easily implemented as a processor interconnection. A graph of valency three might be an acceptable number. To achieve a graph of certain girth a minimum number of nodes must be employed (30). For a graph of valency three the relationship between girth and number of nodes m is

$$m = 2(1+2+2^2+..2^{(r-1)}) \qquad (1)$$

where $r = g/2$ (2)

For example a girth 6 graph requires 14 nodes and a girth 8 graph requires 30 nodes. An increase of 2 for the girth doubles the number of nodes required. It may not be economical to built larger girth machine because of the number of nodes required. It is important for a particular girth the smallest graph should be employed. The task of finding a minimal sized graph for a particular girth is hard, for example a trivalent graph of girth 9 with 58 nodes (36).

1.7 Objectives

The interconnection strategy of Bowyer et. al. forms the starting point for this research. There are several ways in which a network of multiprocessors based on the proposed interconnection strategy can be driven. The reduction machine model was chosen because it offers the possibility of making the multiprocessor general purpose.

There are three main areas of research activities involved in this thesis. Below are the descriptions of each activity.

(i). The simulation studies of the behaviour of the reduction machine on the interprocessor network. A simulator model was developed. The basic structure of the simulator describes the interconnection network. The reduction machine model is built on top of the basic simulator. The techniques experimentally simulated are data flow execution, sparse and regular tree evaluations.

(ii). The next activity was the construction of the multiprocessor. The component processors employed were MC68000 based microcomputers each with 256K memory. These were acquired from an external source. However the interprocessor interface hardware was designed and built by the author. An important requirement for the interface is that it should have a high data transfer rate. The interface chosen was a shared memory which resides between two adjacent processors on the network.

The shared memory is 2K bytes wide and allows bidirectional communication. In order to aid handshaking, interrupt hardware is provided. A complete set of shared memory interfaces was built enabling the construction of a six node trivalent graph network.

(iii). The final activity is the implementation of a run time kernel and demand-pull schedulling for an abstract reduction machine. The run time kernel is responsible for task management and the organisation of communication between processors. A system of data structures records the information of every task created. The information is kept valid until the task is terminated. A task creates parallel subtasks by issuing intructions that are placed on the instruction queue. The instruction can be consumed locally by the host processor or it could migrate onto one of the neighbouring processors. A processor that requires tasks does so by issuing a demand to one of its neigbours. This is done by setting a flag in the shared memory. In this way the migrating of tasks across the network is done by the process of pulling as opposed to the tasks being pushed to the idle processors.

CHAPTER 2 SIMULATION

2.1 Need for simulation

The behaviour of the interprocessor network can be studied using a simulator (7)(11). The information that can be obtained from the exercise are -

1. The speed at which all the processors can be utilised;
2. The average processor utilisation during the computation.

The exercise was done on several parallel evaluation strategies. They are the following -

1. Data flow;
2. Demand driven for sparse tree;
3. Demand driven for regular tree.

From the simulation important design decisions can then be made. The objective of the simulation is to study the load distribution characteristic of the network.

2.2 Model

The simulator can be programmed using a sequential or a concurrent language. The following sections describe the model required for both sequential and concurrent programs.

2.2.1 Parallel Model

Ideally a concurrent language like concurrent Pascal or Ada should be used for programming the simulator. The process or task construct of these languages readily describes the node's activity.

Logical characteristics of the interprocessor communication can be represented by the communication construct. However this is a very simplistic communication model, because the physical behaviour of the interface hardware cannot be modelled accurately. The behaviour of the interface can be modelled more accurately by an intermediate process linking the two processes representing the nodes.

The basic structure of the simulator consists of a system of processes representing the nodes and a system of processes representing the communication interface. In a six nodes system there will be six processes representing the nodes and nine processes representing the interface. In the implementation of the run time system for a concurrent language the effect of parallel process is produced by interleaving the execution. The scheduling of the process is controlled by a clock. If the rate of this clock is sufficiently fast a true parallelism effect can be produced.

The simulation is based on costs that represent the time for computation and the time for communication. Both the computation cost and communication cost are dependent on the size of the problem to be executed. However on top of these costs is the cost incurred by the intrinsic characteristic of the network. It is most likely that this cost is constant. For the purpose of the simulation exercise the costs are represented by numbers whose initial values can be varied. Experiments

were done by assigning various values to the cost and observing the load balancing, distribution and processor utilisation effect on the network.

The cost is simulated by a delay function executed by the processes. The delay function argument is a number. The value of this argument determines the length of the delay.

The computation cost is wholly dependent on the size of the problem. However the cost for communication is not wholly dependent on the size of the imformation going through it. In the real network the communication is asynchronous. With asynchronous communication the response can be affected by a purely random chance. To illustrate this effect let us observe how rendezvous takes place. A sender for the sake of argument, sends out a request for a transfer. The sender will wait an indeterminate length of time for a response from the receiver. The time necessary for both nodes to rendezvous depends on the states the nodes are in. The time taken for a node to get out of its present state in order to rendezvous is not constant. One method of simulating this random event is to incorporate a random number generator in the delay function. The overall delay effect of the delay function will be dependent on the input argument and the random number produced internally.

2.2.2 Sequential Model

It is possible to describe the simulator using sequential language. However the accuracy of the model will be less than that of a model described by a concurrent language. In the concurrent run time system the interleaving is done at a fine grain level of the host processor. Using a sequential language the lowest level of interleaving possible is at statement level. However this can be messy to implememt. A possible description of n processes running in parallel but without communication is as below -

```
Repeat
  process1
  process2
   .
   .
  processn
Until ..
```

The delaying effect can be realised as follows. The argument to the process is initially set to some value. On entering the process this number is decremented. If this number is non zero no further action is done. The parallelism effect is preserved because the rate the processes are interleaved is equivalent to interleaving at instruction level. The processes from 1 to n are identical to each other. When there is more than one

class of identical processes it may not be clear where to put the other set of processes. For example the processes can be interleaved as -

process1a
process1b
.
.
process2a
process2b
.
.

The alternative is to group the processes.

process1a
process2a
.
.
process1b
process2b

The 'b' sets of processes may be dependent on the 'a' sets of processes. For example if process 'b' is the communication process, it will only be invoked when process 'a' wishes to use the communication facilities.

The logical interconnection can be set up in a table. The nodes can be referred to by numbers, for a six nodes system from one to six. Next a table of six

rows and six column is defined. In the entry for a corresponding row and column is a value of 0,1,2 and 3. A value zero signifies that there is no connection between the corresponding row and column node. The values 1,2 and 3 signify the port used to established the connection.

2.2.3 Form adopted

At the time the simulation exercise was carried out there was no concurrent programming system available locally. The objective of the exercise is not so much at getting a precise result as obtaining a general feel for how the network would behave for different parallel evaluation strategies. Results obtained from an approximate description by a sequential language should be adequate and hence this form was chosen. The following sections describe the various experiments using the simulator. For the different parallel evaluator appropriate systems are built on top of the basic simulator.

2.3 Data Flow machine

A data flow machine of section 1.4.1 can be built on top of the basic simulator of the previous section. Recapitulating, the data flow machine mechanisms are the fetch unit, arithmetic unit and update unit. In addition there are the data structures corresponding to the instruction queue and activity store. According to

the discussion in the previous section, the mechanism must be treated as processes that represent all the activities on a node. A fetching process involves fetching instructions from the local instruction queue or from one of the neighbouring nodes' queues. Fetching instructions from neighbours is a communication process. The activity store is treated as a unified global memory space. There is no hardware global memory. Each node is allocated a certain range in the virtual memory space. If a memory reference falls on non local allocated space, it assumes that the system automatically issues the request through the communication interface. For the purpose of simulation, the activity store is a global array. The distinction between a local access and non local access is that it will take longer to serve a non local access. This can be simulated easily in the update process.

The activity store is an array of activity templates or records. The fields of this record as already described in the last chapter are the instruction, data receptors a and b, and destination. The instruction queues contain pointers to active templates.

The simulator should be able to simulate networks of various nodes without extensive modification of the program. Since the activities of the various processes in the simulator are identical for all the nodes, the program can be table driven. Variables are stored in an

array. When processing a particular node, the node identifier is used as the index to refer to the appropriate variable. The only data that needs to be changed when simulating different sized networks is the node size and the interconnection table.

In the simulator there is a subsidiary part of the program which generates a binary tree of activity templates. The tree generator accepts an argument which specifies the depth of the tree. The leaves' activity templates by definition are active and fireable. The addresses of these templates are places on the instruction queue of a node chosen as the central site. An alternative to initial loading a central node is to evenly load all the nodes.

The operation of the simulator is as follows. The fetch function attempts to get instructions from the local queue. If there are no instructions available locally, the fetch process will try to steal an instruction from the neighbours. If an instruction is available from one of the neighbours, no further attempt will be made to obtain instructions from the other two neighbours. After the fetch processes for all the nodes have been executed, the next step is the computation process. The result from the computation is use to update the destination template. If the destination template is fireable, the address of the template is placed in its host node. The destination node is identified from the address of the template.

The time dependent characteristics of the various mechanisms of a data flow machine were ignored. The combined operations of fetch, computation and update are performed in every simulation cycle. Effectively the simulator developed is a synchronous parallel data flow machine.

2.3.1 Results and discussion

Some measurements were performed on a 14 node network. A binary tree of activity templates of specific size is built in the activity store. The addresses of activity templates at leaf level were distributed evenly onto the queues of all nodes. The processor utilisation against time is shown in fig 2.1 and fig 2.2. Looking at the first graph, all the nodes are active for the first three cycles. The number of active nodes began to drop gradually to two active nodes. The computation progressed for a further four cycles before decreasing to one and terminating. The general shape of the second graph is similar to the first, but the initial maximum utilisation of nodes remains longer. This indicates there are large number of active activity templates available locally. As the computation progress towards the root, the number of activity templates is halved with every level. Theoretically, towards the end of computation the number of active nodes should reduced by half for every machine cycle. From the second graph, the tail off is

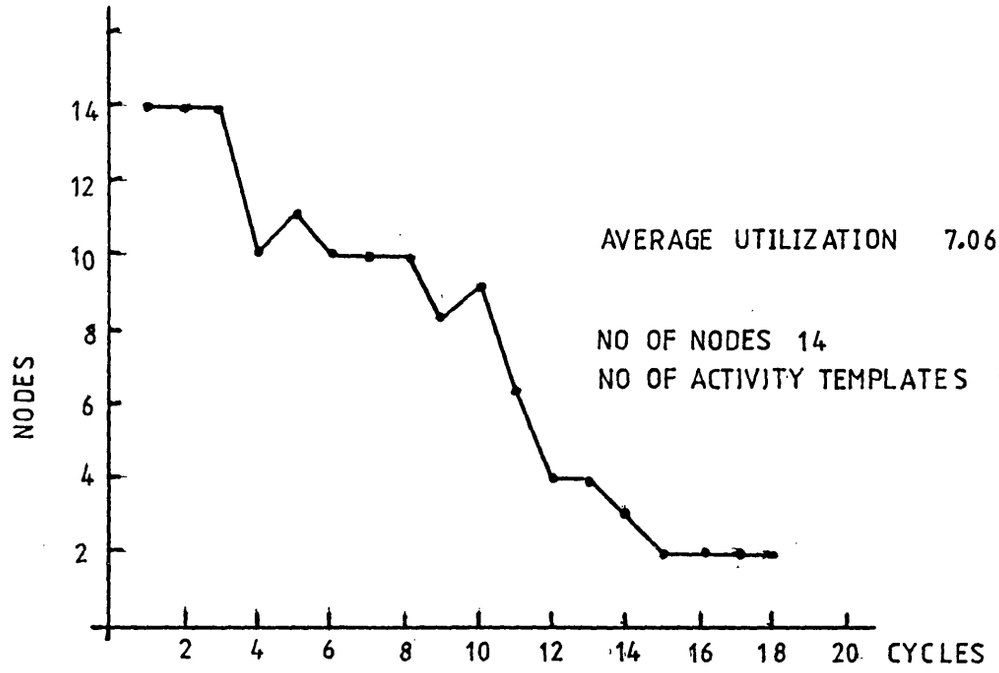


FIG. 2.1 DATA FLOW RESULT 1

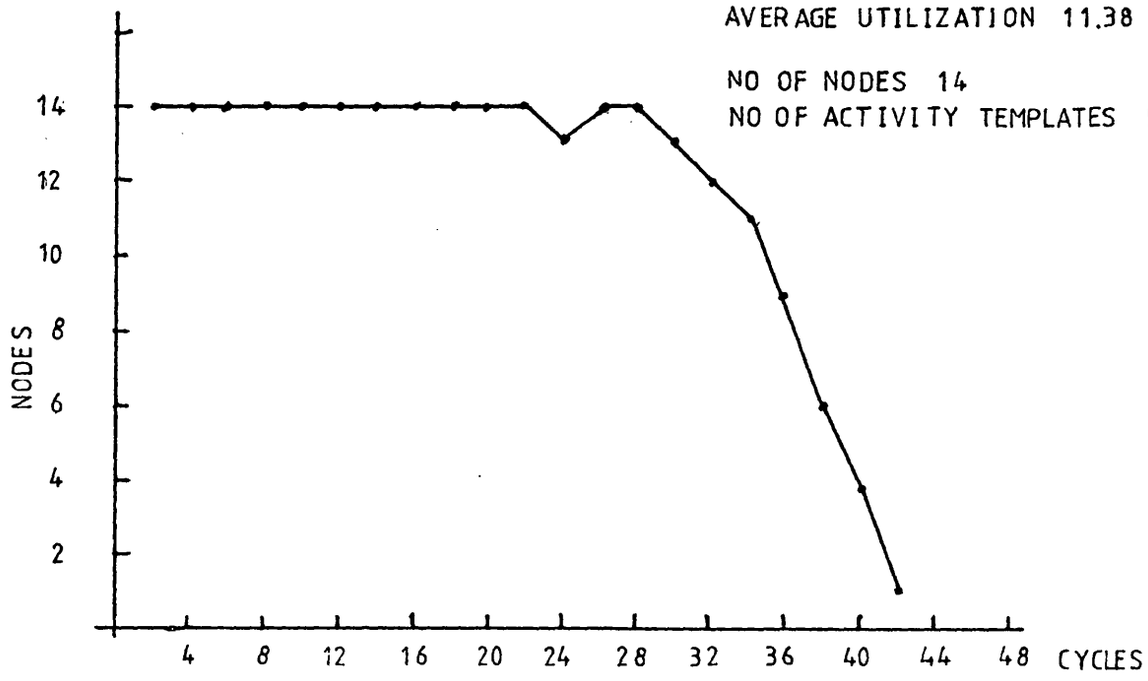


FIG. 2.2 DATA FLOW RESULT 2

at a rate of one node for every machine cycle. The reason for this is that the instructions for the remaining active activity templates are not evenly distributed. If the instruction templates are localised on a few nodes, the rate at which the idle nodes can grab the instructions are low. This is because the busy nodes can service the request for instructions at predefine point of the simulator cycle.

Below are results of average processor utilisation against activity templates size. Average utilisation is defined as the number of activity templates divided by the number of machine cycles.

object size	127	512
average utilisation	7.06	11.38

A simple conclusion that can be drawn from this experiment is:

> When the object size is significantly larger than the number of nodes the average utilisation is high.

2.4 Demand driven machine

In a way the structure of a demand driven machine shares some of the mechanism of data driven machine. However the generation of the flow graph is done at run time. An initial instruction is placed on the instruction queue of a central node which is chosen

arbitrarily. The instruction is fetched by the host node. Depending on the size of the problem which the instruction represents, the node will attempt to subdivide this problem. To sustain parallel execution, at least two instructions must be produced. The instructions generated are placed on the local instruction queue. The local node has first priority to the instruction. After the first instruction on the queue has been retrieved, the remaining instruction will be available to the neighbouring nodes. In a normal divide and conquer evaluation, the problem is recursively evaluated until the leaf computations are reached. When this instant is reached a system of flow graph similar to the data flow program tree has been built. The computation can be stopped at the point where the leaf computations are reached. This is when the results demanded in the computation are the leaf computations. The computation can be made to proceed further by the leaves passing results to their parents. The unwinding process continues until results from subproblems reach the initial parent problem.

2.4.1 Sparse tree evaluation

Quicksort (31)(33) is a fast array sorting algorithm and can represent a suitable indicator for testing the performance of an architecture. A brief description of the quicksort algorithm is as follows: An item x in the array is picked up at random. The

array is scanned from the left until an item ai>x is found. The array is now scanned from the right until an item aj<x is found. The two items are then swapped. The scan and swap process is continued until the two scans meet. The array is now partitioned with the left part having items less than x and the right part having items greater than x. The partitioning process is repeated on both parts of the array and so on recursively. Figure 2.3 is a quicksort program which describes the algorithm using recursion and is written in Pascal. The program is due to Wirth (32). The two statements -

```
if l<j then sort(l,j);
if i<r then sort(i,r)
```

determine whether further partitioning is necessary.

```
program quicksort;
   procedure sort (l,r: index);
      var i,j: index; x,w: item;
   begin i :=l; j := r;
      x := a[(l+r) div 2];
      repeat
        while a[i].key < x.key do i := i+1;
        while x.key < a[j].key do j := j-1;
        if i <= j then
        begin w := a[i]; a[i] := a[j]; a[j] := w;
        i := i+1; j := j-1
        end
```

```
        until i > j;
        if l < j then sort(l,j);
        if i < r then sort(i,r)
    end ;
begin sort(1,n)
end
```

figure 2.3

For example if l<j is false the left part is not partioned. However i<r can still be true. Therefore only the right part is partitioned. The result is the evaluation tree is not regular.

The time taken by each partioning step is dependent on the size of the array and also a random probability. The simulation can be described as follows:

Starting Q(n)

after α_n time

[where α is a random variable and $\alpha \in (1,n)$]

Splits into

Q(β)

Q(n-β-1)

where $\beta \in (1,n-1)$ and Q(1) $\Rightarrow \emptyset$

2.4.2 Results and discussion

Experiments were done on quicksort. The results presented here are for a 14 nodes machine. Two different array size were fed into the simulator.

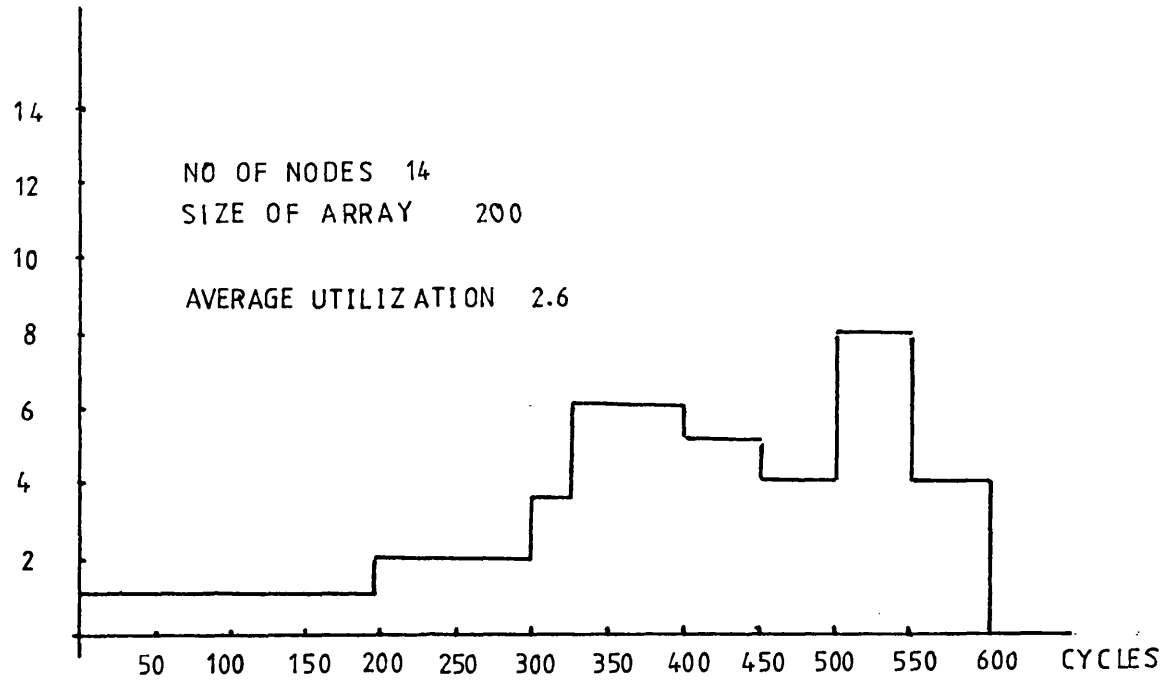


FIG. 2.3 QUICKSORT RESULT 1

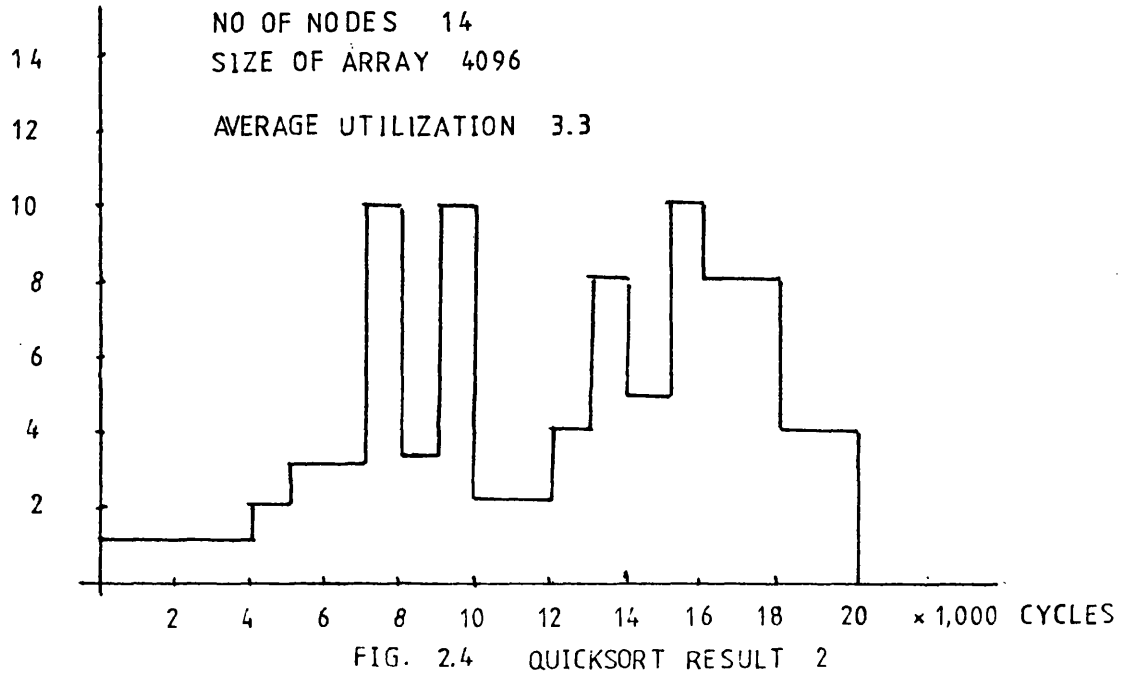


FIG. 2.4 QUICKSORT RESULT 2

Figures 2.4 and 2.5 are the processor utilisation graphs for the experiment. Average utilisation is defined as the total number of executions divided by the total machine cycles. It can be seen that the processor utilisation is very low - 2.6. With the larger problem, the utilisation is only slightly better - 3.3. An improvement with a larger size problem is expected simply because a larger size problem should offer a higher degree of parallelism. The results do not show that quicksort will work well on the architecture. It seems that the generation of tasks is not fast enough to sustain parallelism. When two tasks are created as a result of a partition, the lives of the two tasks may not be equal. Out of two tasks it is expected that four more tasks are created. If this occurs at the same time, four nodes can work at the same time. However, if the four tasks are created one after another, it suffices to have only one node for doing the job.

2.5 Regular tree evaluation

A Fibonacci function is defined as

$$f(n) = f(n-1) + f(n-2) \quad \text{for } n > 2$$

$$f(0) = 1$$

$$f(1) = 1.$$

This function is capable of creating two invocations of itself whenever it is called. The potential concurrency is high as the structure generated by the function is a

regular binary tree.

One structure built into the simulator skeleton is a queue that is used to hold the instruction which is responsible for creating the function. The argument to the function is a number N. The function recursively generates two further functions with arguments N-1 and N-2. The function is suspended while waiting for the results of the children functions to come back. In order to be able to reactivate a suspended function, a descriptor record is set up for every function invoked. In the simulation, the parameters that were varied are:

1. N - the function's argument
2. Cn - number of cycles required for computation.

2.5.1 Results and discussion

Figures 2.6 to 2.8 are the processor utilisation during computation against machine cycles for the Fibonacci's number experiment. The three graphs were obtained for different computation times Cn. The graphs exhibit a general shape that indicate heavy computation occurs during the second quarter of the total machine cycles.

The efficiency is defined as-

$$e = \frac{(\text{total splitting time}) + (\text{total computing time}) * 100}{\text{total machine cycles} * \text{number of nodes}}$$

The efficiency tends to increase with higher N.

N=10 Cn=1 - e=35%

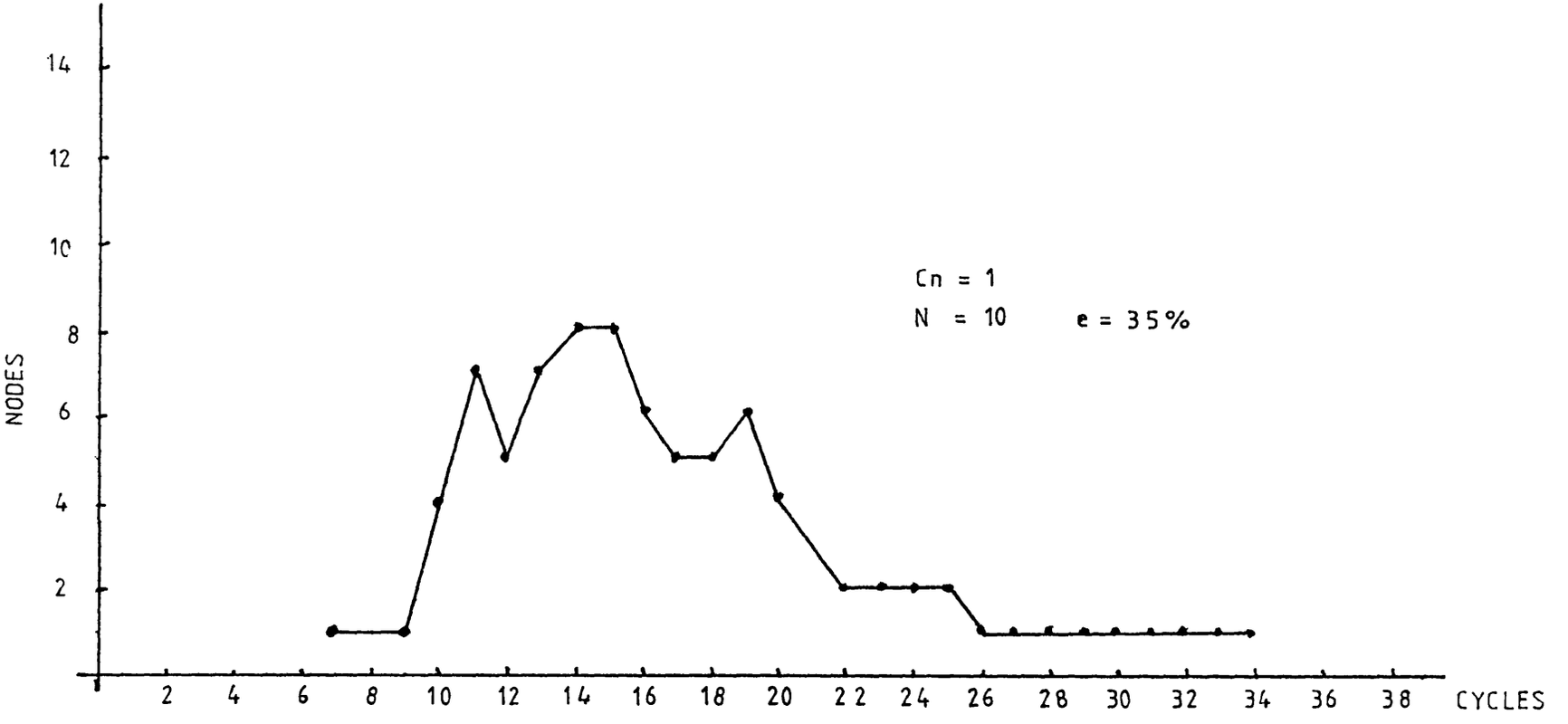


FIG. 2.5 FIBONACCI RESULT 1

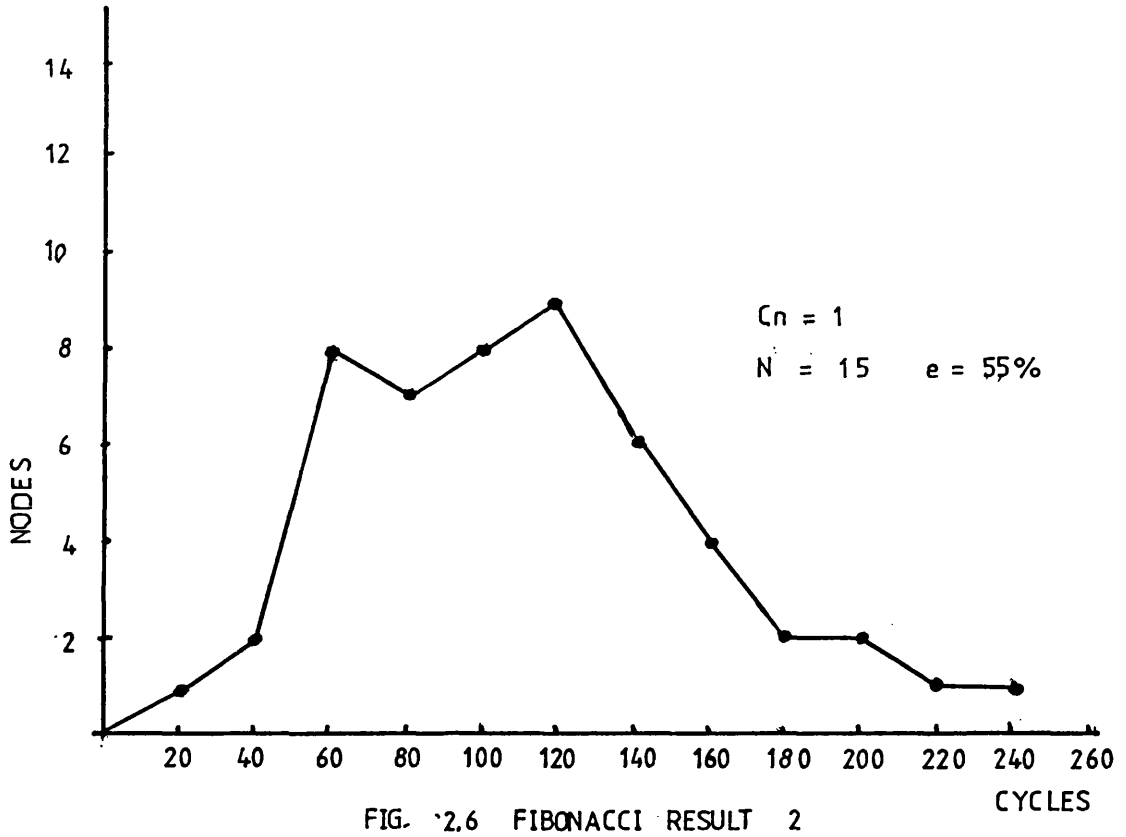


FIG. 2.6 FIBONACCI RESULT 2

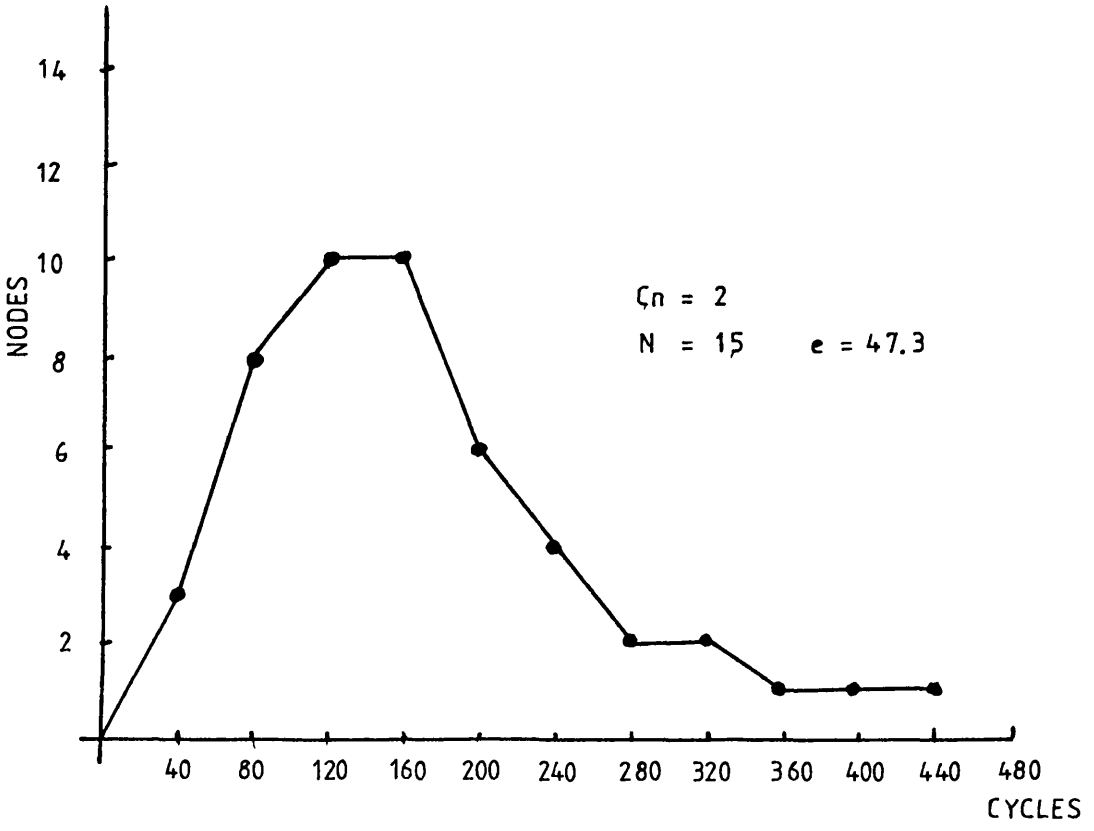


FIG. 2.7 FIBONACCI RESULT 3

N=15 Cn=1 - e=55%

N=15 Cn=2 - e=47.3%

For N=15 there is a drop in efficiency with Cn=2. The result obtained from the simulation shows that an efficiency of up to 55% is possible. This is considerably better than that was obtained for quicksort. A conclusion that can be derived from the two simulations is the evaluation tree for the computation must be regular and the availability of large numbers of parallel tasks if the architecture is to realise its potential parallelism.

2.6 Conclusion

The simulations were done on three methods of driving parallel computers. The methods are data flow, demand driven evaluation on sparse tree (quicksort) and demand driven evaluation on regular tree (Fibonnaci's number). The best result was obtained for the data flow followed by the Fibonacci's number and lastly the quicksort. A possible reason why data flow is faster compared to demand driven is the evaluation tree is already set up in the data flow machine prior to the machine starting up.

For any computer network it is impossible tp achieve the idealised efficiency of 100%. The factors that reduced efficiency are

1. overhead associated with communication;
2. distribution of tasks onto processors.

The distribution of tasks can be affected by the nature of the problem and the characteristic of the network configuration.

In the experiments the efficiency increases with larger problem size. For the problems simulated it can be concluded that the tasks distribution is problem bound rather than network bound. Since the tasks distribution is problem bound, the use of λ^{q} more complicated interprocessor network would not improve the result. The trivalent graph of maximal girth network minimised the number of nodes required for a given girth, therefore it should be an economical system for implementing multiprocessors.

CHAPTER 3 HARDWARE

3.1 Introduction

Having decided on the abstract architecture, it was next desirable to identify a suitable node processor. It was considered essential to choose one of the newer microprocessors, to allow testing of substantial problems. This had to be one which could provide with a basic software environment.

3.2 Choice of processor

The choice of MC68000 microprocessors as node processors was made for the following reasons. The MC68000 supports high level languages efficiently due to its consistent architecture, large number of registers, large addressing range and special high level oriented type of instructions. The MC68000 has a total of seventeen 32 bits registers in addition to the 32 bits program counter and 16 bits status register. The address bus is 24 bits wide and the data bus is 16 bits wide. Implementing an operating system is made easy by the availability of priviledged instructions, memory management and a multi level interrupt and trap structure. The MC68000 was designed to support multiprocessing. Both hardware and software interlocks are provided for multiprocessor systems. Bus arbitration logic is provided to handle access contention in shared bus or shared memory environments. The software interlock is provided by the special instruction (TAS - test and set operands).

It would have been beyond the scope of the project if the node microprocessor system had been built from scratch. The best choice is to obtain board level computers of the type normally supplied to the OEM (original equipment manufacturer) market. A system that suited this requirement was obtained from the School of Electrical Engineering. The system comprises of a four card set mounted in a cage. The cards making the set are the MC68000 processor, rom card, 256K ram card and the input output and front panel display card. The card size is double eurocard. Six such systems were employed for the multiprocessor system. The cages carrying the individual microcomputer system are mounted in an instrument rack. To realise the multiprocessor system a set of communication hardware was designed and built by the author. Subsequent text in this chapter describes the design, implementation and testing of the interface between processors.

3.3 Choice of communication interface

The communication interface can have a considerable effect on the performance of the multiprocessor system. A global shared memory, although capable of modelling any logical interconnection scheme, is not suitable due to the contention problem. A true high speed interface can be provided by direct memory access hardware controlling parallel data lines. However the pure efficiency of a hardware scheme is not the only

criterion that has to be considered. The cost constraint is a major factor that affects any design decision. Another factor is the physical constraint. Indirectly or directly the physical constraint is related to the cost. The cost can be kept down if all the interface hardware required for each processor can be built on one card comprising of three seperate interfaces. Each interface links one processor to another processor.

3.3.1 Programmed control or DMA

The cheapest type of interface can be provided by serial lines, but serial communication under program control is too slow. Parallel communication under program control is considerably faster and may satisfy the speed requirement. Both methods of data transfer can be made very fast by having direct memory access control. However the circuitry of a direct memory access controller is somewhat complex. To implement a direct memory access controller using standard TTL devices requires an enormous chip count. The circuitry can be implemented using a VLSI direct memory access controller for MC68000. However this device was not available at the time the interface hardware was designed. The decision not to pursue a DMA controlled interface is partly due to this logistic situation. In controlling the interface hardware there will be intervention by software to some extent: even in DMA

communication where most of the difficult tasks have been tackled by the hardware, the setting up of the device is done by the program. It can be assumed that under most circumstances the receiving processor is always busy at the instant the transmitting processor initiates a data transfer. In order to set up the DMA hardware ready for reception the receiving processor has to be interrupted from its current processing state. If this request cannot be granted instantly the transmitting node will be held up momentarily. This situation does not occur if there is a buffer in between the transmitting and the receiving node. The buffer forms a pipeline. In a way this provides some degree of parallelism.

3.3.2 Buffer memory

Global shared memory was rejected initially because of problems with memory contention, but if a shared memory is only shared between two processors there should not be any noticeable degradation in effeciency. The worst case memory access time is twice the time for a normal access and this occurs when both processors are reading or writing to the memory simultaneously. The transfer rate possible by this kind of shared memory is better than that possible by direct parallel communication under program control. In such program controlled parallel communication, handshaking is required for every word transfered. Handshaking can

be expensive in processing time as it involves polling and setting of a protocol flag. Considerable saving in handshake processing can be achieved if handshaking is only done for every block of data transferred.

3.3.3 Polling or interrupt

However some form of signalling is needed for handshaking at the block level. The MC68000 Test and Set instruction can be used to implement semaphore logic. This is one solution but it relies on polling. The disadvantage of polling is that the polling processor is continually accessing the shared memory. It will be more efficient if access to the shared memory is only for actual data transfer. Interrupts seems to be a better solution but at the expense of additional interrupt hardware. The MC68000 provides seven levels of interrupt: this is enough to implement the handshake interrupt hierarchy. It is foreseeable that the maximum number of interrupts required is two levels. One level is required for signalling a request and the other level for acknowledgement. The interrupt vector can be supplied by the hardware or generated automatically in the autovector mode.

3.4 The solution adopted

The dual ported shared memory based on the discussion and arguments presented above was chosen for the interface. At this stage the practicality of

putting three dual ported memories and the interrupt circuitry on one card was still unknown. An attempt was made to design the circuit with minimum chip count and, by careful layout, all the circuits fitted on one double size eurocard wirewrapping board. The normal method to reduce chip count in a hardware design is to use VLSI chips. Because of the specialised nature of the circuitry this is not possible within the constraint of the project. The circuit must be built entirely using standard TTL device with the exception of the memory device.

3.5 Board space and connectors

For a six node system there are nine interconnection paths. Logically the shared memory is midpoint between two nodes(fig. 3.1). Translating this physically, the shared memory resides on a stand alone card connected to the two processors by two sets of cables. Cables from the processors cannot simply emanate from the bus: the signals going to the cables must be buffered. A card for buffers and cable connectors is required in every cage. In all there will be fifteen cards needed. The number of cards can be reduced to six if the memory card is hosted by one processor. The other processor only holds the buffer card. In terms of cost, there will be considerable savings because the other set of buffers is no longer required.

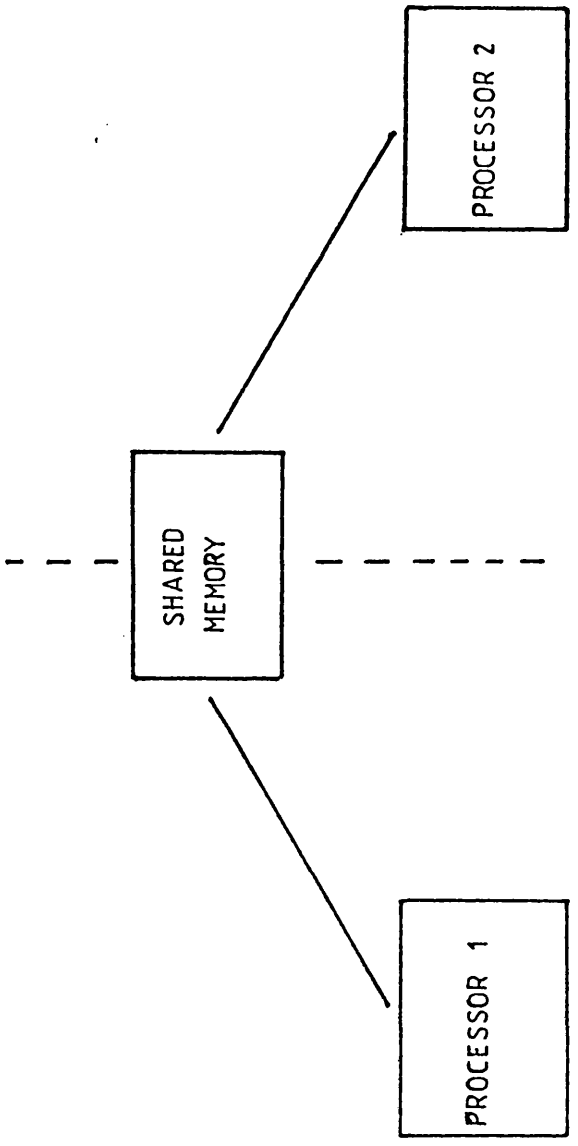


FIGURE 3.1 LOGICAL POSITION OF SHARED MEMORY

However this creates organisation problems. On which processors to place the memory cards and on which processors to place the buffer cards?. The cards can be organised as follows. The memory cards and buffer cards are placed on alternate processors. Figure 3.2 shows a six node system. This diagram shows that the scheme is feasible. To handle any future expansion the card arrangement will work for larger node sizes. By means of graphical exercises it was discovered that the arrangement fits graphs with even girth and with an even number of nodes. The reason for this is as follows. To establish an interconnection both types of cards are required. It is not difficult to see that if the girth length is not divisible by two the remainder represents an extra buffer or memory. The arrangement will not work for the four node girth 3 graph and Petersen's graph of ten nodes with girth 5. This limitation should not be a major problem.

3.6 Design considerations

The limit decided earlier was to build a complete set of communication interfaces on six boards. Three of the boards must accommodate three shared memories and the interrupt controller. The other three boards accommodate three buffers and the interrupt controllers. In addition both types of board require further associated circuitries which are an address decoder and an input port for control purpose. To build

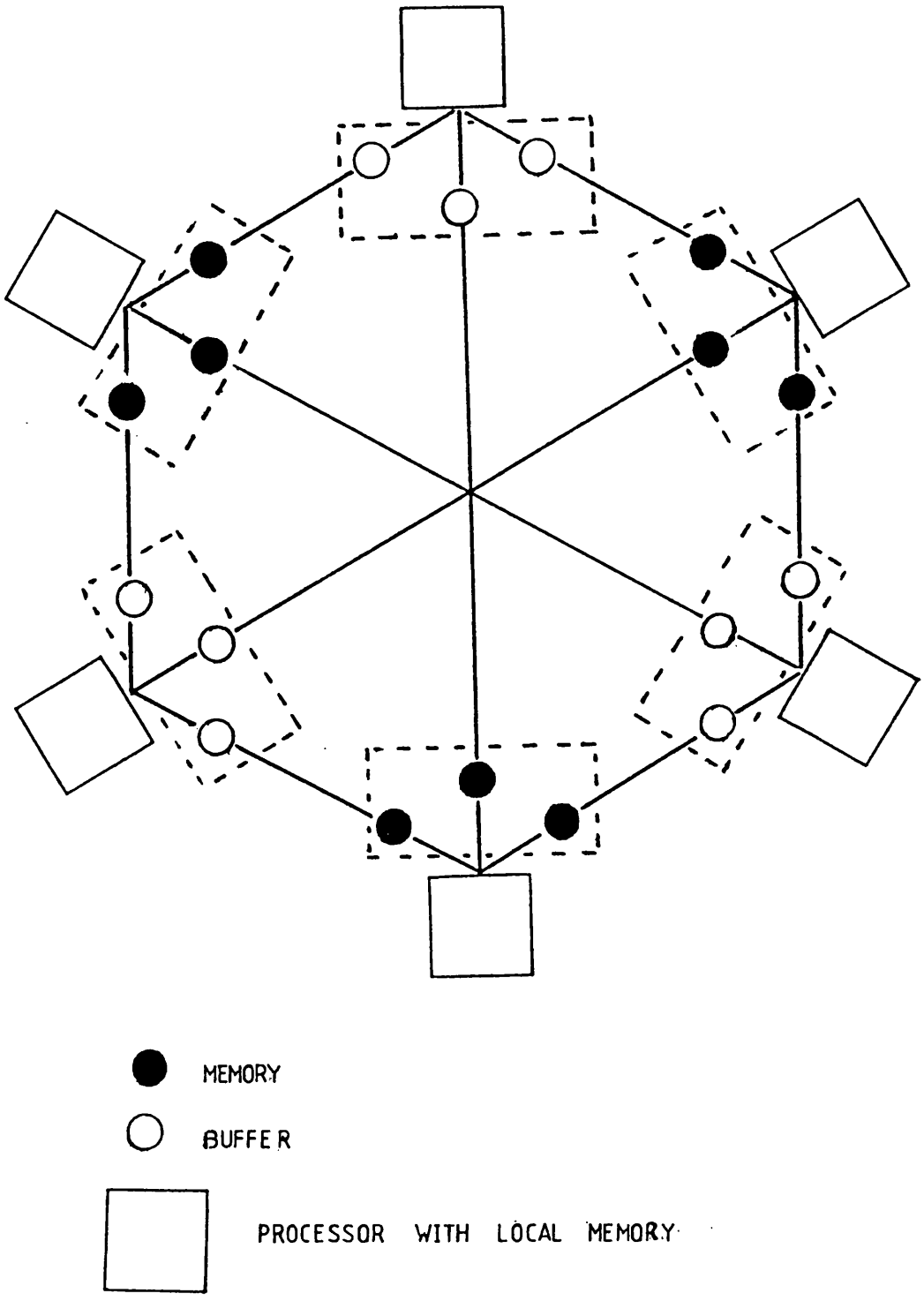


FIGURE 3.2 INTERFACE CARD ORGANISATION

the first type of board can pose a problem. It may not be possible to build all the circuits on one board.

A size of two Kbyte should be ample for the shared memory. Consequently eleven address lines and sixteen data lines are required. Connection between interface boards is by forty way ribbon cable. Each board contain three forty way connectors. The connectors occupy some board space. This has to be taken into consideration when designing the circuits.

In order to produce the design for the two ported memory and the interrupt hardware it is imperative that the functions of the MC68000 signals are fully understood. The information on the MC68000 is obtained from two Motorola publications, the MC68000 user's manual (58) and the MC68000 data sheet (59). The description of the MC68000 signals and bus operations are given in appendix A.

3.7 Two ported shared memory

Viewed from either port the shared memory looks no different from ordinary memory. The existence of another processor hooked on the opposite port should not interfere with the operations of the first processor. The actual RAM device in the memory is a shared resource. The addresses and data from both processors cannot be applied simultaneosly on the RAM. In this situation one of the processors must be blocked from accessing the RAM until the first processor has

terminated its access cycle. The blocking should be done transparently. The MC68000 asynchronous transfer mode helps in the design of the shared memory. During a blocked access the processor will treat the memory as a slow device. The shared memory consists of a RAM memory device and a controller circuitry. The memory controller coordinates the requests from the processors. The memory controller serves as the interface between the RAM and the MC68000 signals. It is therefore essential for the memory controller to interpret the MC68000 bus operations correctly.

Figure 3.3 shows the various components of the memory. Control signals from the processors are fed to the memory controller. The controller outputs signals that control the RAM buffers and issues $\overline{\text{DTACK}}$. The controller has to perform arbitration when there is a simultaneous request. The operations of the arbiter can be quite complex. One condition that must be avoided at all cost is the race hazard due to the processors being totally asynchronous to each other. When the term "simultaneous request" is used it is supposed to encompass the following:

1. the difference between the time of arrival of the first processor's request and of the second processor's request is zero;
2. the difference is finite but less than the length of an access cycle.

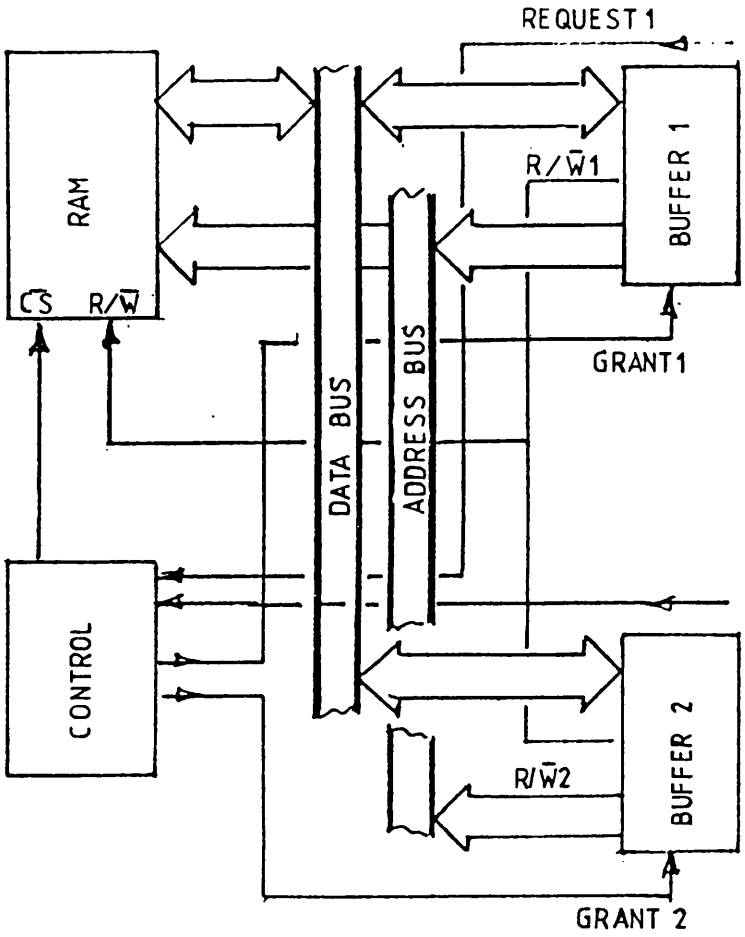


FIG.3.3 A TWO PORTED MEMORY

An arbiter hardware takes a finite time to make a decision. There will not be any problems with the first case. However with the second case, the late arrival of the second request can upset the working of the arbiter. The problem of race hazard can be partially eliminated by adopting synchronous hardware design. In synchronous design both requests will be sampled by a clock and thus eliminate the second case effect above. All the functions of the memory controller can be realised using a state machine.

3.7.1 State machine arbiter

The state machine can be designed using discrete logic, ROM or programmable logic array(PLA). However it is not feasible with discrete logic as this approach uses a large amount of chips. The choice is between PLA and ROM based machines. PLA devices are generally more expensive than ROM, thus a ROM based machine was selected.

The shared memory is selected by decoding its address and qualifying it with address strobe($\overline{AS}$). The state machine recognises this as a request. Figure 3.4 shows the state diagram for processing the request. The machine cycles through state 0 awaiting a request. When a simultaneous request occurs it is logical to assume that request 1 has higher priority than request 2. The state machine has to generate $\overline{DTACK}$ and various buffer and RAM control signals in response to the request.

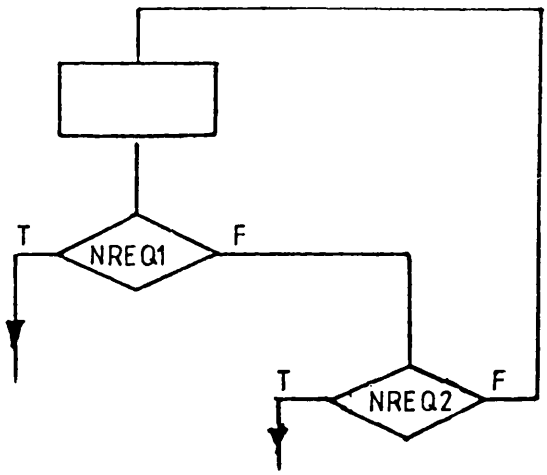


FIG. 3.4 ARBITRATION

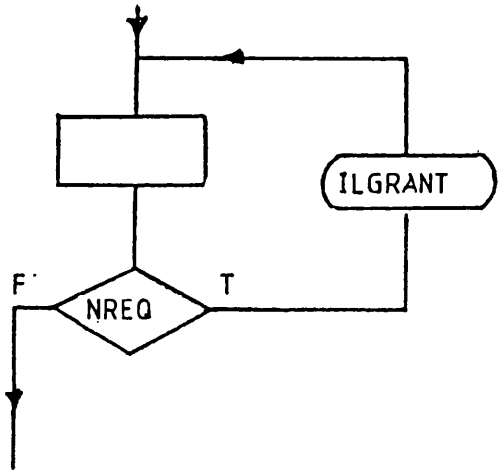


FIG. 3.5 WAIT LOOP

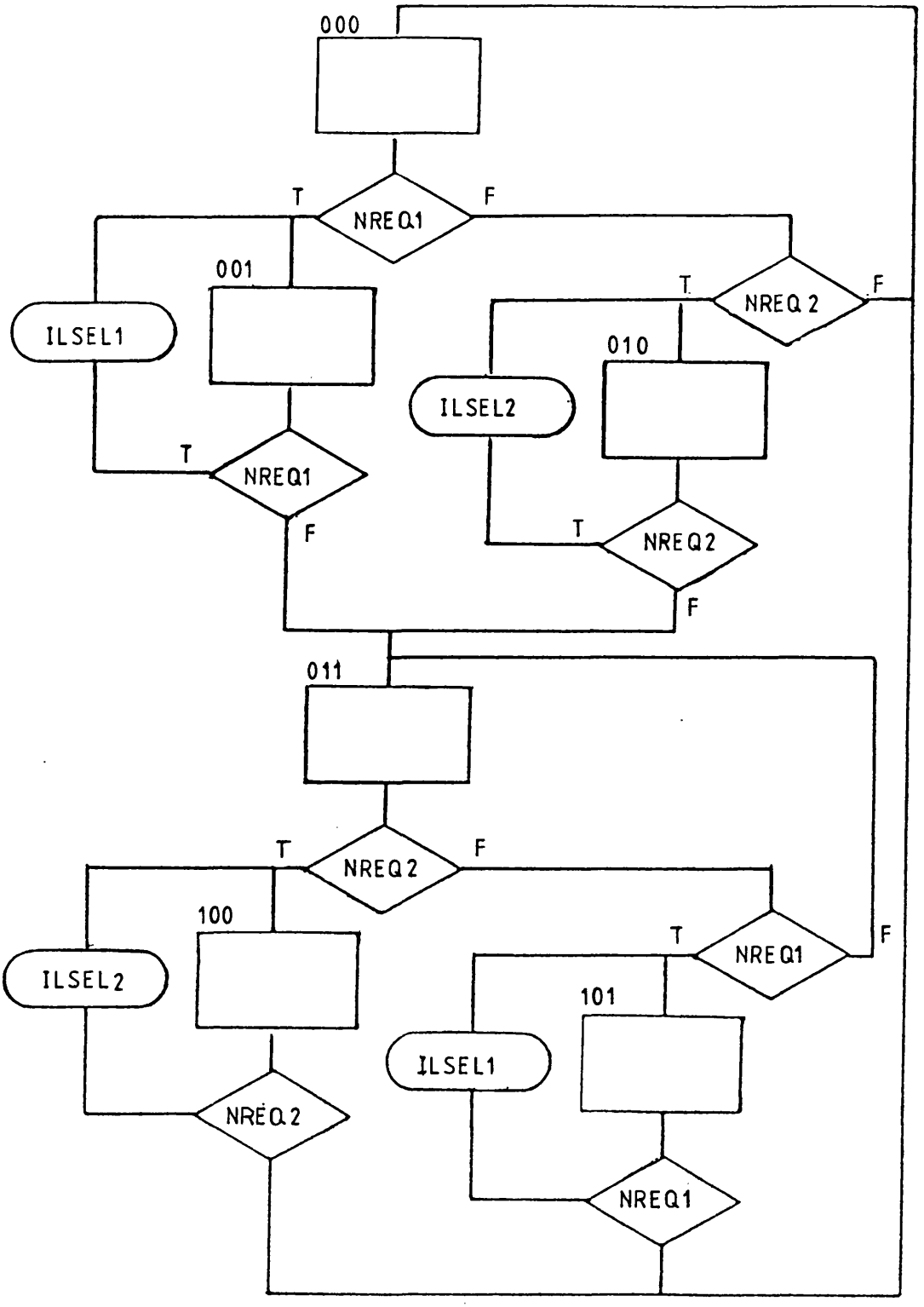


FIG.3.6 STATE DIAGRAM FOR MEMORY ARBITER

	NREQ1	NREQ2	A	B	C	NA	NB	NC	ILSEL1	ILSEL2
L1	0	x	0	0	0	0	0	1	1	1
L2	1	0	0	0	0	0	1	0	1	1
L3	1	1	0	0	0	0	0	0	1	1
L4	0	x	0	0	1	0	0	1	0	1
L5	1	x	0	0	1	0	1	1	1	1
L6	x	0	0	1	0	0	1	0	1	0
L7	x	1	0	1	0	0	1	1	1	1
L8	x	0	0	1	1	1	0	0	1	1
L9	x	0	1	0	0	1	0	0	1	0
L10	x	1	1	0	0	0	0	0	1	1
L11	0	1	0	1	1	1	0	1	1	1
L12	1	1	0	1	1	0	1	1	1	1
L13	0	x	1	0	1	1	0	1	0	1
L14	1	x	1	0	1	0	0	0	1	1

Fig. 3.7 State table for arbiter

DTACK must be issued before the end of state S4 of the processor if a wait state is to be avoided. This can be achieved by deriving DTACK from the conditional output (LSEL1 or LSEL2). DTACK is held low for as long as address request (NREQ1 or NREQ2) is asserted. Figure 3.5 shows a wait loop for generating $\overline{\text{DTACK}}$ while address request is held asserted. The same signal is used for enabling the RAM. To give the second processor the same chance as the first processor at getting control of the memory, the priority of NREQ1 and NREQ2 should be rotated. Figure 3.6 shows the complete state diagram and figure 3.7 shows the state assignment table for the machine. The state table can be accommodated in a 32 by 5 ROM.

The state machine is constructed using a fast bipolar rom (TBP18030) and a 74LS175 latch. The PROM has a finite lookup time of the order of 40ns. The state clock is derived from one of the processor's clocks running at 8 Mhz. The combine propagation delay of the state latch and the ROM access time must therefore be less than 125 ns. The state machine is synchronised to one of the processor's bus operation. The bus operation of the other processor is totally asynchronous. There is a chance of the ROM giving false output when input changes.

The result is glitches produced at the ROM outputs. Since the LSEL1 and LSEL2 are conditional outputs they cannot be buffered by the state

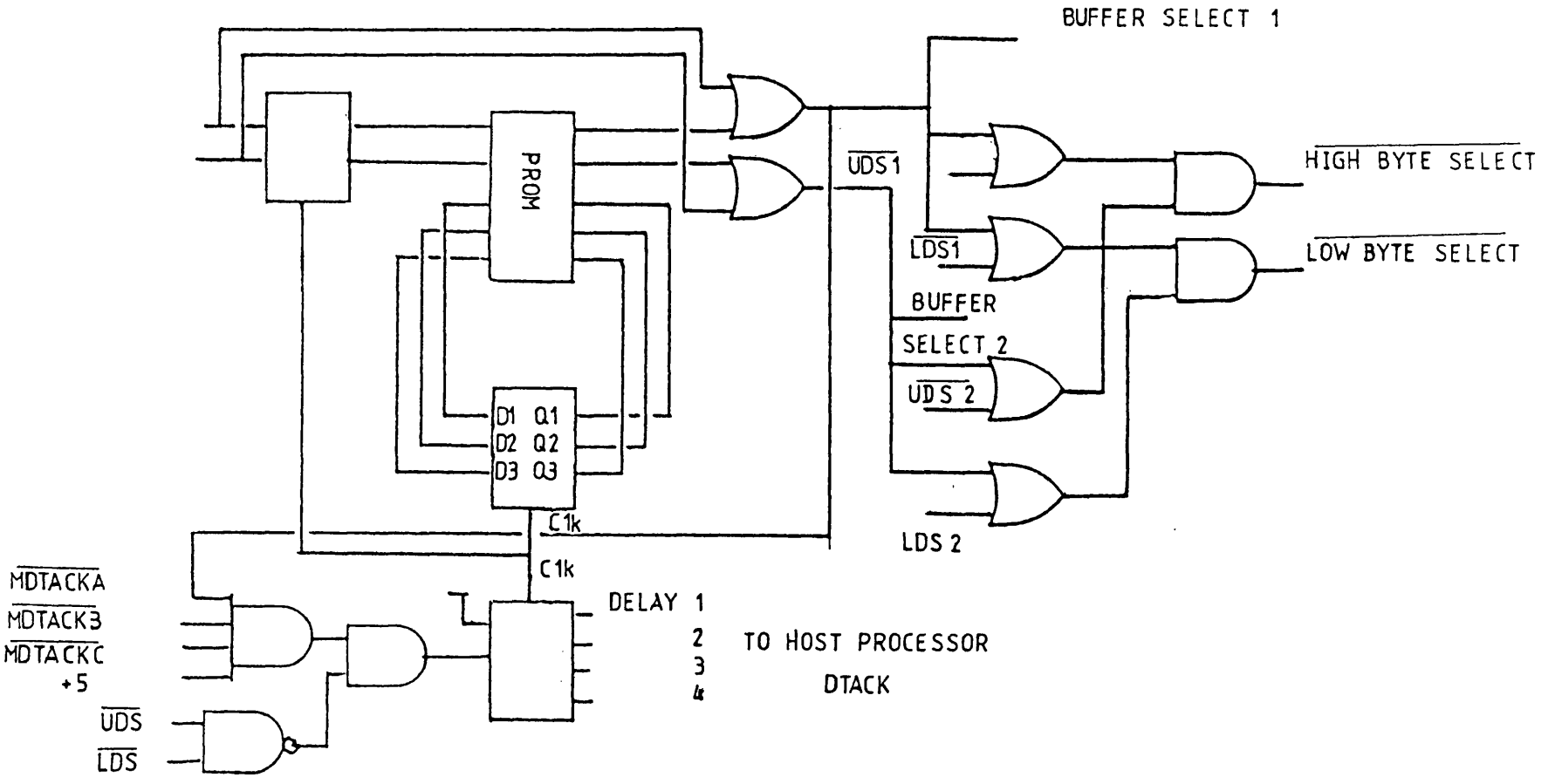


FIG. 3.8 CONTROL CIRCUIT

transition. It is therefore neccessary to buffer the request inputs. But this inadvertently upset the timing in relation to the MC68000. The request will be delayed by one clock period. This means $\overline{\text{DTACK}}$ cannot be issued before the end of state S4. The addition of one wait state is not disastrous. The more damaging effect is that $\overline{\text{DTACK}}$ cannot be negated before state S0 of the next cycle. However this problem can be solved by gating the output by the non delayed memory request input. 'Oring' the request input with the output will bring the signal high as soon as the input is brought high (fig. 3.8).

3.7.2 Byte addressing considerations

In order to allow byte access, the memory has to use two RAM devices, one RAM for the low byte and another RAM for the high byte. The upper and lower data strobes gated with the controller signal are used to select the appropriate RAM. For the RAM, two MK4118-4 static rams from Hitachi are used. For proper operation, the ram set up and hold time must be observed. Although the MC68000 timing would have tackled the characteristic of static RAM, the additions of address and data buffers can upset hold and set up time of the ram. Therefore the gating of the control signals to the RAM is critical.

The MK4118-4 is a 1 K by 8 device. There are ten address inputs and eight data lines. There are four

control lines, write enable ($\overline{WE}$), chip select ($\overline{CS}$), output enable ($\overline{OE}$) and latch input signal (L). The function of the latch input signal (L) is to determine whether the mode of operation is asynchronous or synchronous. In asynchronous mode where L is high, the MK4118-4 provides a fast address ripple through access of data. In synchronous mode a transition of L from high to low will latch the address and the $\overline{CS}$ inputs. In the design the asynchronous mode of operation is adopted.

The select output from the state machine controls the enabling of the address and data buffer, chip select and gating of the write enable of the RAM. The read write signal is routed through a tri-state buffer(fig. 3.9). The buffered read write signals from both processors are wired-ored together. This signal is gated by the upper data strobe $\overline{UDS}$ for the upper RAM and gated by the lower data strobe $\overline{LDS}$ for the lower RAM.

In a read operation data will be valid after a period of 250ns at the maximum for the MK4118-4. This parameter is the address access time tAA. The data will be valid for a maximum period of 125ns. This parameter is the chip select data off time tCSZ.

In a write operation, the write cycle is initiated by the $\overline{WE}$ pulled low. The $\overline{CS}$ must also be low. The $\overline{CS}$ is always low by the time $\overline{WE}$ enable is applied because of the propagation delay through the OR gate. Data is

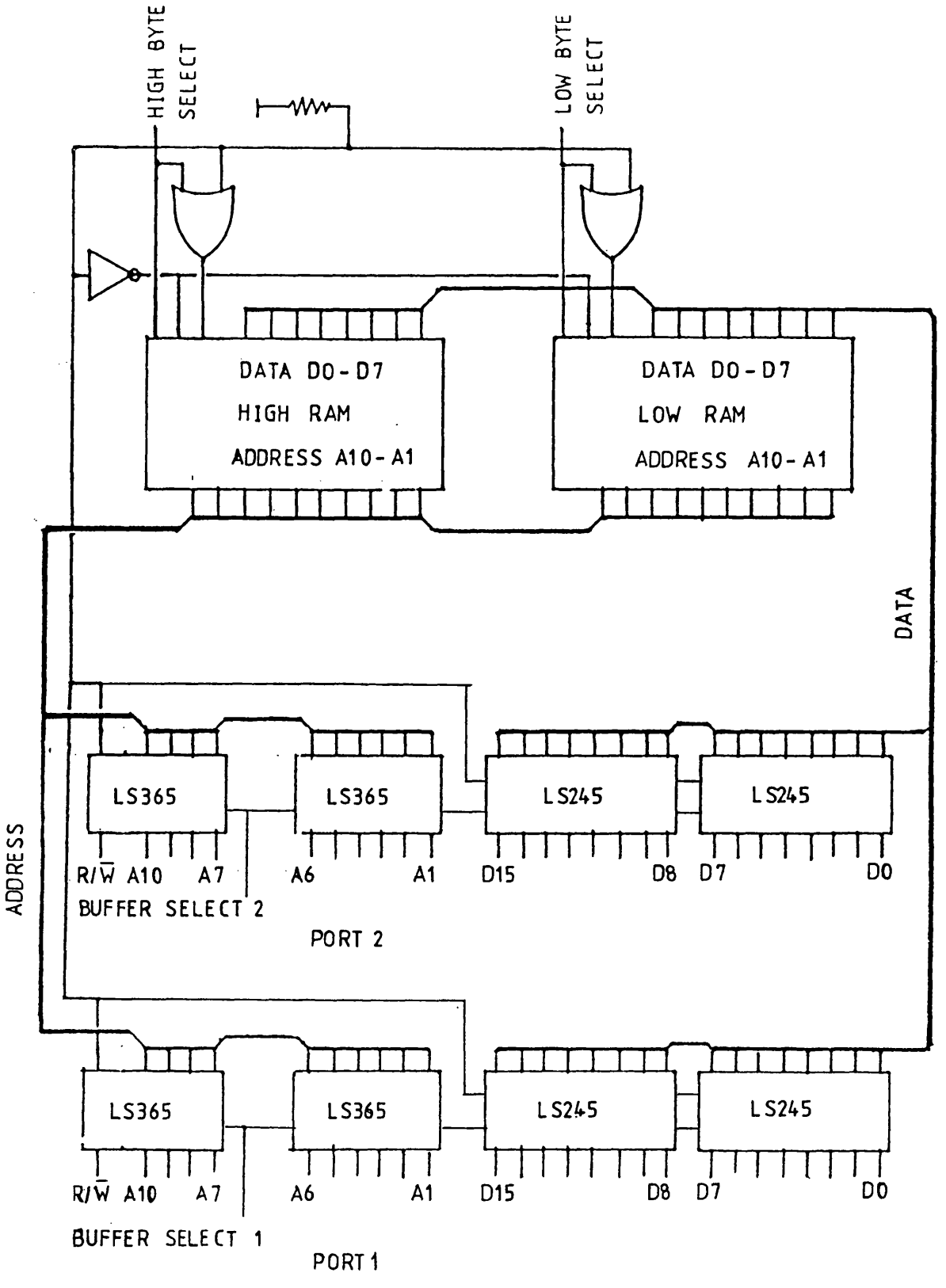


FIG. 3.9 TWO PORTED MEMORY

written into the RAM on the positive transition of the $\overline{WE}$ signal. The data must be valid for a certain duration prior to the transition and also the data must be stable for a certain duration after the transition. These parameters are known as data to write set up time(tDSW) and data from write hold time(tDHW). The set up time(tDSW) and data hold time(tDHW) for the MK4118-4 are both 50ns. To guarantee the hold time, the WE signal must go up before the data buffer is disabled. The chip select $\overline{CS}$ and $\overline{WE}$ signals to the RAM are brought high when the data strobe signal($\overline{LDS}$ or $\overline{UDS}$) is negated. There are three gate levels through which the data strobe passes before reaching the $\overline{WE}$ pin of the RAM. The approximate propagation delay is 50ns. The buffer select signal from the state machine is 'ored' with the memory request signal. The memory request signal is qualified by address strobe($\overline{AS}$). In decoding the address the address strobe($\overline{AS}$) is introduced early in the decoding chain(sec 3.11, fig. 3.19). There is a long propagation delay from the transition of address strobe($\overline{AS}$). The approximate propagation time through five gate levels is 80ns. The data must be held stable at least 50ns after $\overline{WE}$ is negated. The data buffer must take approximately 20ns for it to be disabled in order to satisfy this requirement. It is recognised that introducing delay by relying on propagation delay of chips is not a good design practice. Two proper methods of introducing delay are employing delay line and using

a clocked D type latch. It is not feasible to use a clocked latch because delay can only be achieved for integral clock periods. The reason for using gate propagation delay as opposed to delay line is chiefly that it is cheaper to employ the former method.

The processor data transfer acknowledge $\overline{\text{DTACK}}$ should not be asserted directly by the select output. This is to allow for the access time of the RAM. In the read operation, the data output will be valid 120ns after chip select is pulled low. For a successful read operation for the processor, data must be valid at the latest 90ns after $\overline{\text{DTACK}}$ is asserted. Therefore it is necessary to delay $\overline{\text{DTACK}}$. The delay required is 30ns. This can certainly be obtained by adding two redundant gate levels. However to allow for a wide safety margin, the delay is provided by a shift register. Delaying DTACK has no useful effect in the write operations. The necessary set up(tDSW) and data hold times(tDHW) have already been met.

3.8 Buffers

The memory card is designed to slot in one of the processors' card cage. The other processor is linked to the memory by a ribbon cable. The address, data and control bus are buffered on the processor card, but the buffering is designed to cope with the mothercard loading only. The bus must be buffered first before driving the cable. The passive cards do not carry the

memory but a set of driver and buffer chips. On the memory card, the port linked to the remote processor must also be able to drive the cable. On the memory card the buffers serve two functions. The first is to isolate the two ports and the second is to drive the heavy load caused by the cable. The maximum length of the cable used is about 30 inches. At this length the capacative effect between adjacent wires can be considerable (18). Cross talk will be a major problem. The step that can be taken to reduce this problem is to use ribbon cable with alternate ground wires.

3.9 Interrupt processing

It has been mentioned previously that two types of interrupt are required. They are 'ready to transmit interrupt' and 'acknowledgement interrupt'. The acknowledgement interrupt is at a higher level of priority. Since communication is bidirectional a communication path requires both types of interrupt at each end. At each interrupt level there are three sources, one from each communication link. The MC68000 provides the capability to process seven interrupt levels; ample since only two are required. The task of recognising the source of interrupt at a particular level is left to a combination of additional hardware and software.

The interrupt controller can either supply the interrupt vector or use the autovector mode. At each

level there can be three sources where the interrupt can originate. Possibly there will be three separate routines to service the interrupt, one routine for every interrupt line. The interrupt controller can generate three different vectors corresponding to the three interrupt sources. The processing of the interrupt can then commence immediately. In the autovector mode, any request on the three interrupt lines will cause the processor to jump to the same vector. The task of identifying the interrupt must be done by software will take longer than if provided by hardware. In the design of the interrupt controller both possibilities were explored. The following sections describe the designs.

3.9.1 Interrupt controller with vector generation

Treated as a black box the interrupt controller has three interrupt lines:input, vector data output and associated processor control lines(fig. 3.10). The controller cannot be pure combinatorial logic because it has to arbitrate between the request lines and to generate signals to the processor in the correct sequence. Therefore the interrupt controller requires a state machine to perform the intelligent function. In a way the state machine is similar to the one employed in the two ported memory. Figure 3.11 shows the section that does the arbitration except that it has three

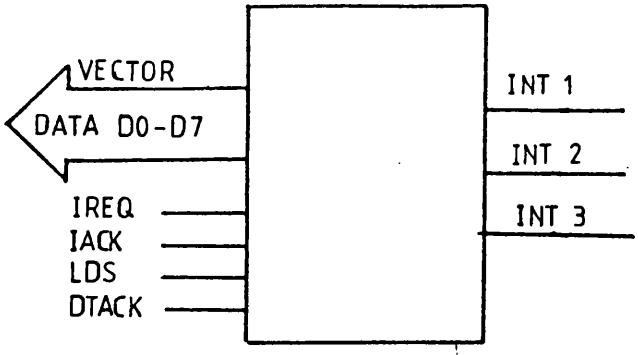


FIGURE 3.10 INTERRUPT CONTROLLER

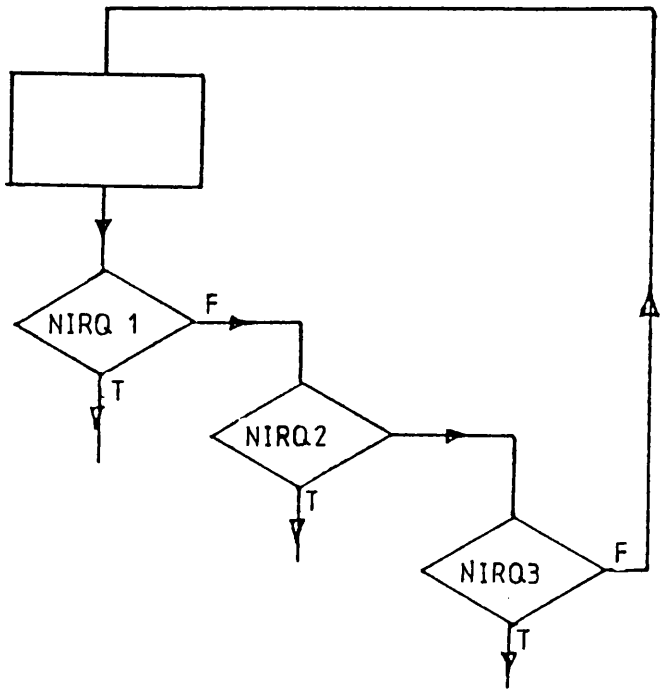


FIGURE 3.11 INTERRUPT REQUEST ARBITRATION

inputs. This is the same technique used to arbitrate in the two ported memory. The outputs from the state machine are the signals that generate the interrupt request to the processor, $\overline{\text{DTACK}}$ and the vectors. The inputs are the interrupt request lines and lower data strobe ($\overline{\text{LDS}}$) interrupt acknowledge ($\overline{\text{IACK}}$). The interrupt request signal coming from the neighbouring node is a single pulse. This pulse must be captured and this is done by a latch. The latch that corresponds to the interrupt line that is being serve must be cleared. Three clear lines must be provided. For the vector data, eight lines are required. In all there will be fifteen outputs in addition to the number of bits required for the state. A minimum of three eight bit wide proms must be used. For practical purposes it is not justifiable to use three proms because large proms tend to be expensive. The number of outputs from the state machine can be reduced by using discrete logic to generate the vectors. The clear lines and vectors can be generated by extra logic controlled by just two outputs from the state machine. The two output lines from the state machine generate the following codes:00,01,10,11. These lines are decoded by a two to four decoder. A '00' output indicates no activity thus the corresponding output from the decoder is unused. The remaining three decoder outputs are used to enable a set of three tristate buffers. The input to the tristate buffer is hardwired to indicate a vector

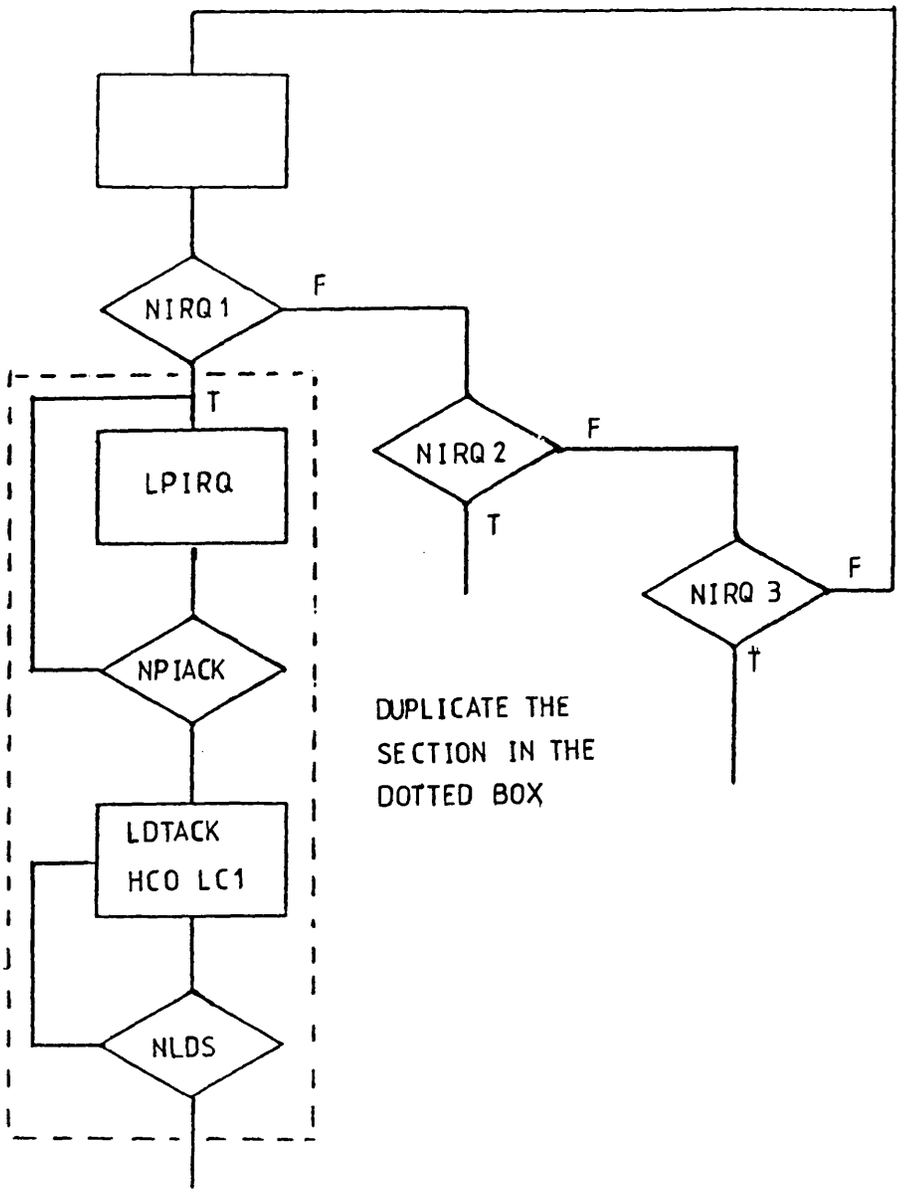


FIG. 3.12 STATE DIAGRAM FOR INTERRUPT CONTROLLER

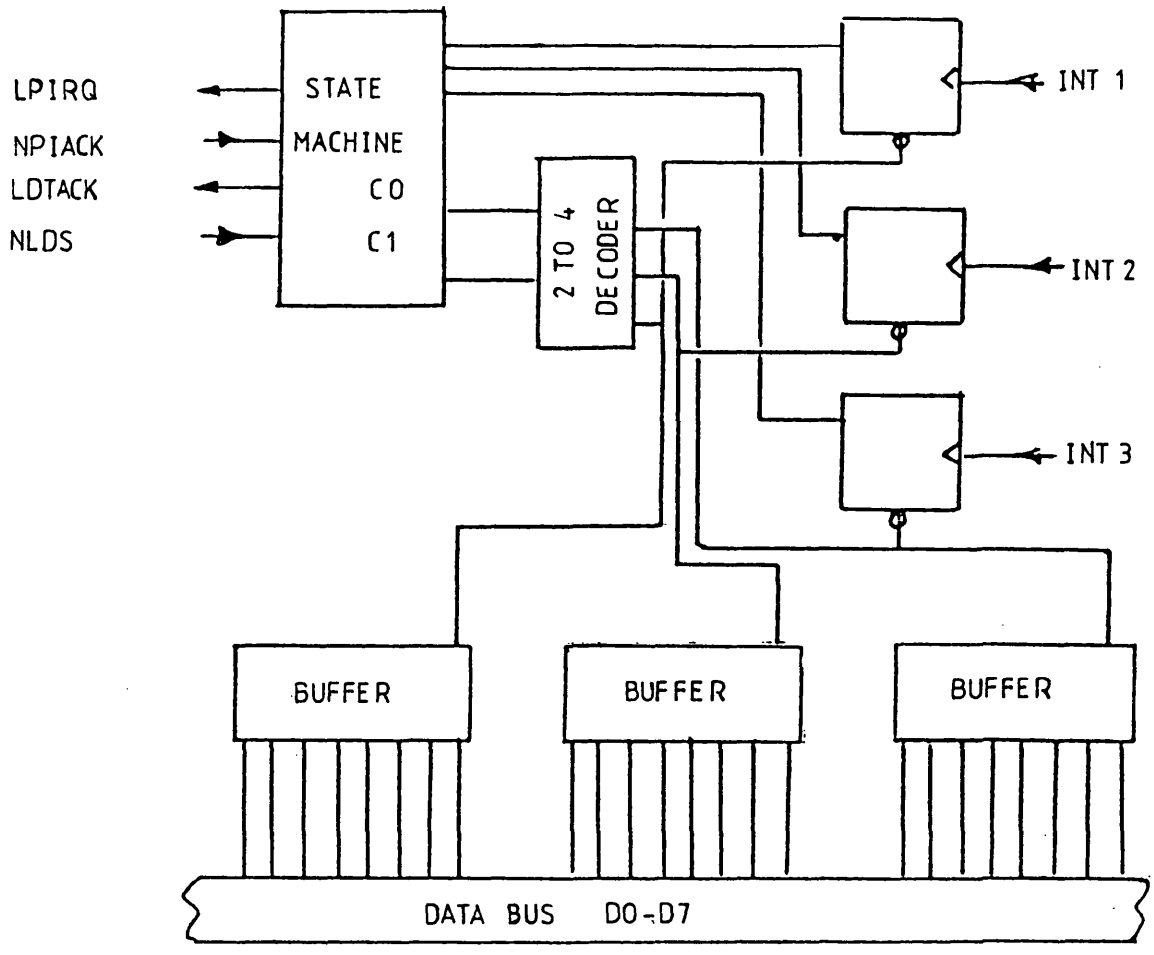


FIG. 3.13 INTERRUPT CONTROLLER WITH VECTOR GENERATION

number. The same decoder outputs are also use to clear the latches. Figure 3.12 shows the state diagram for the controller.

The operation of the interrupt controller in relations to the processor is as follows:
Upon recognising an interrupt request the controller asserts an interrupt to the processor. The controller waits for an interrupt acknowledge signal from the processor. The controller then outputs $\overline{DTACK}$ and generates the interrupt select code. The controller holds the output valid until the processor negates $\overline{LDS}$.

To maintain symmetry an interrupt request line must not maintain an exclusive priority over the other lines. To do this requires a relatively large state machine. Each set of processing sequence in the state machine has to be replicated three times for the order of the request lines rotate. Clearly this is unsatisfactory because a large prom is required.

A second design was arrived at which can reduce the number of states. The design employs a simpler arbitration scheme that decides randomly. The state machine cyclically goes through the state 00,01,10. At each state it outputs a two bit code(00,01,10). A two to four decoder is used to decode this outputs. Each of the outputs is ored with an interrupt request latch. The outputs of the or gates are brought to a three input 'and' gate. The output of the 'and' gate is brought to the state machine. This single input signals

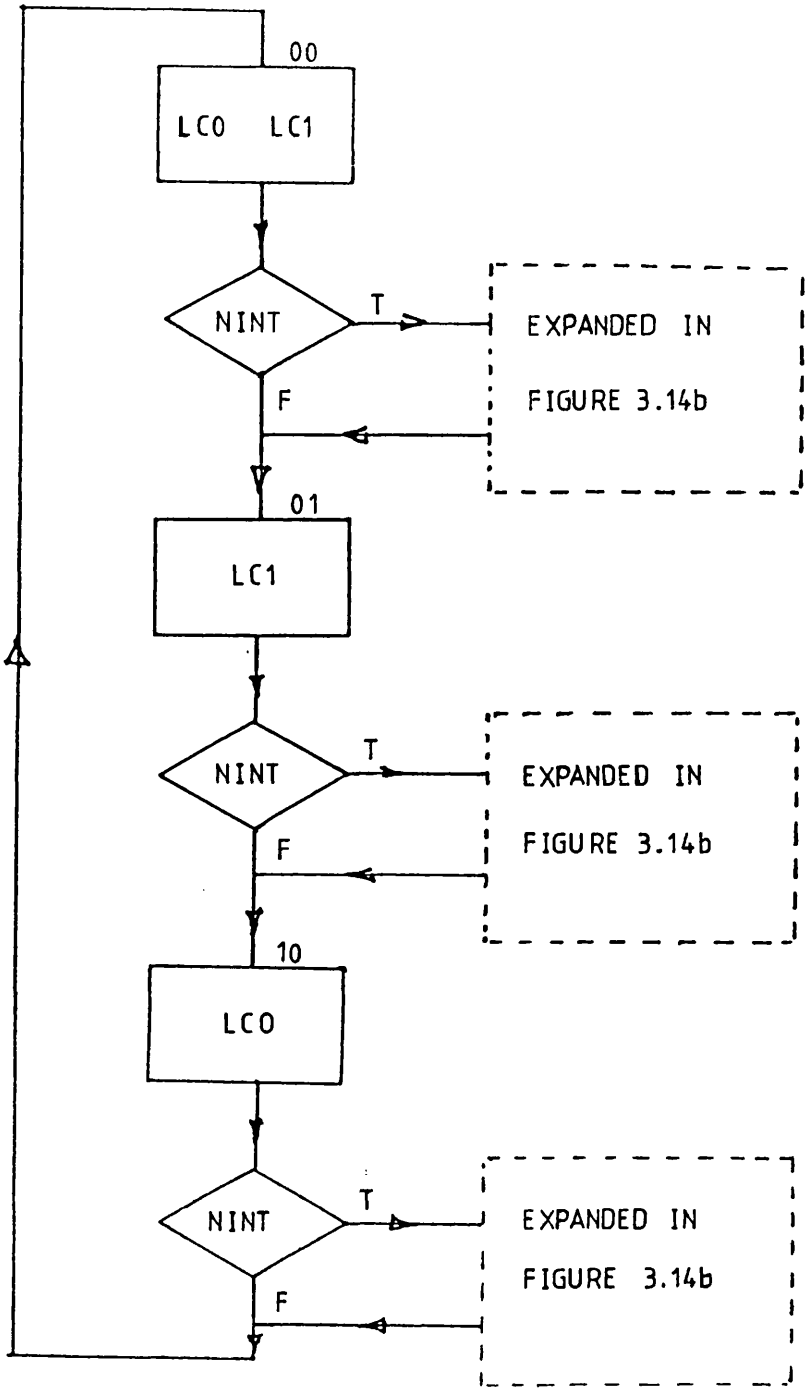


FIGURE 3.14a STATE DIAGRAM FOR MODIFIED INTERRUPT CONTROLLER WITH VECTOR GENERATION

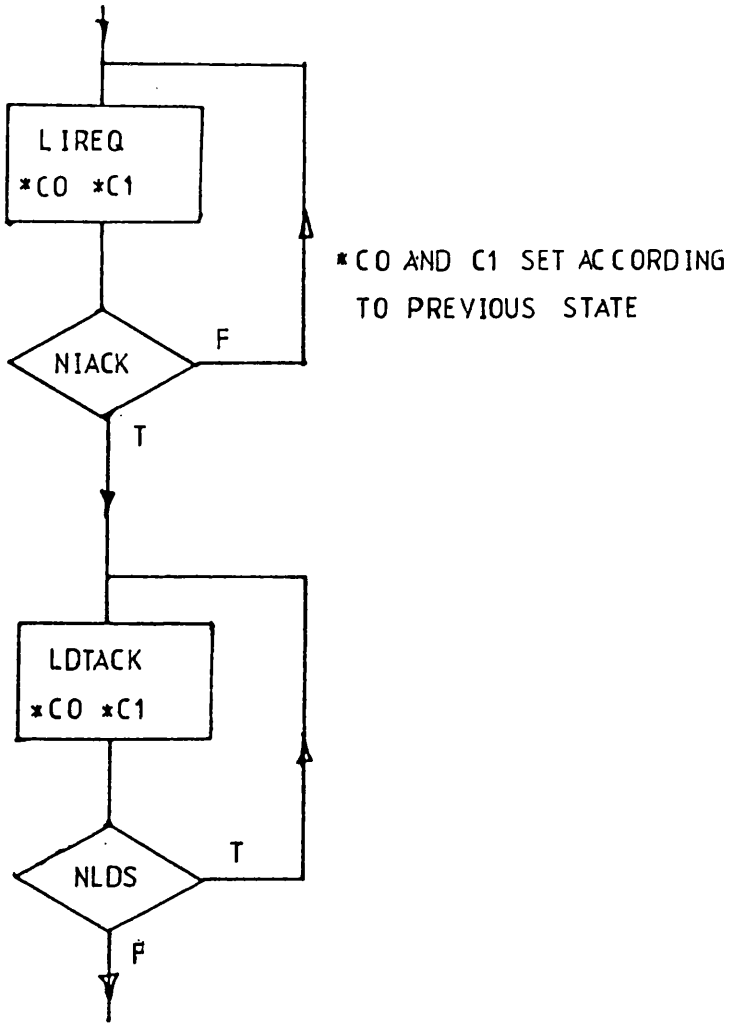

LIREQ
*C0 *C1
NIACK
F
T
LDTACK
*C0 *C1
NLDS
T
F
*C0 AND C1 SET ACCORDING
TO PREVIOUS STATE

FIG.3.14b

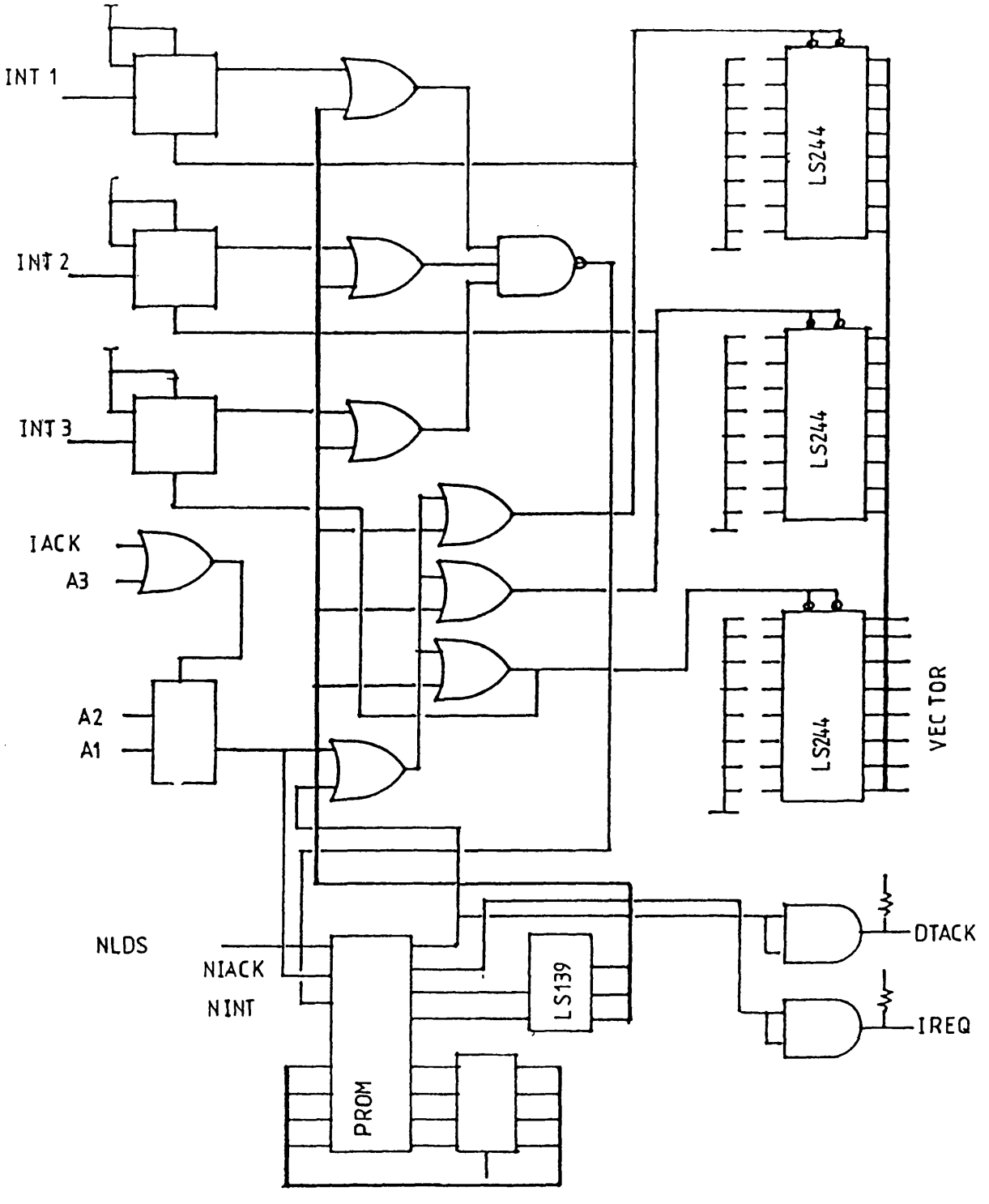


FIG. 3.15 INTERRUPT CONTROLLER WITH VECTOR GENERATION

to the state machine that an interrupt has occured. From this single input it is not possible to identify the source of the interrupt, but the state machine can identify the source of the interrupt by its present state. On recognising the request, the controller then goes into the sequence of issuing the relevant signal to the processor. The relationship between the controller and the processor with regard to issuing an interrupt request, recognising an acknowledgement and generating the vector is similar to the design previously discussed.

Figure 3.14 shows the state diagram for the modified interrupt controller.

3.9.2 Interrupt controller with autovector

The interrupt controller with autovector is simpler and can be built entirely using combinatorial logic. An incoming interrupt will be captured by the latches. The outputs of the latches are 'anded' together by a three input 'and' gate. If any of the latches is low, a low logic signal is generated at the output of the 'and' gate and this is used to signal an interrupt request to the processor. The processor responds by issuing interrupt aknowledge $\overline{\text{IACK}}$ and the interrupt priority level on A1,A2 and A3. The controller circuitry generates valid peripheral $\overline{\text{VPA}}$ signal by the decoding the required priority level when $\overline{\text{IACK}}$ is asserted. The processor has the capability to read the status of the

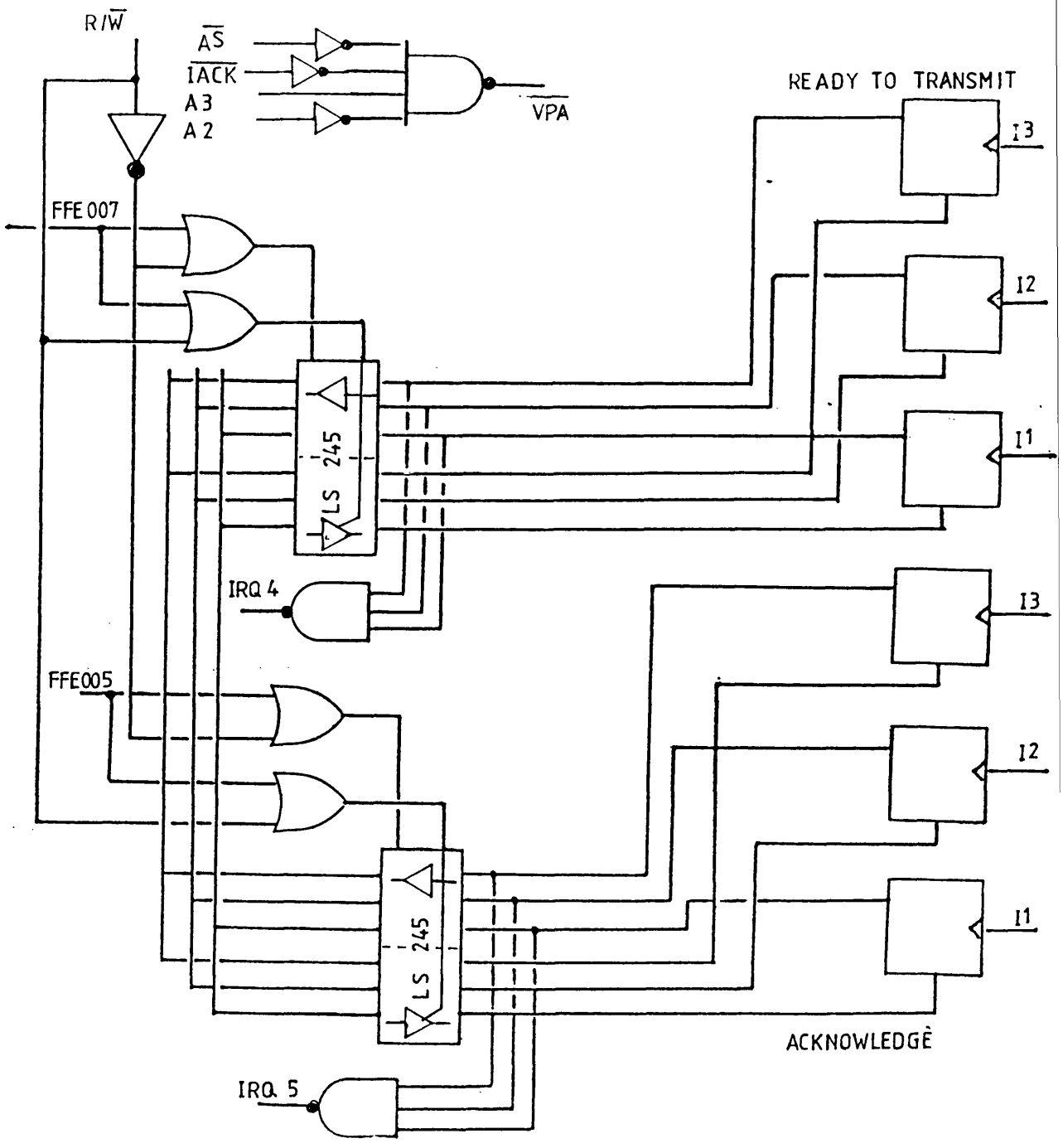


FIG. 3.16 AUTOVECTOR INTERRUPT CONTROLLER

latches. At the same time the processor can clear the interrupt request on the latches. In the event of multiple interrupt requests, the software will decide the interrupt line to service. Only the latch that corresponds to the interrupted line is cleared.

Each interrupt is required to serve three request lines, a total of six request lines for both levels of interrupt. A data input port or data output port can be realised by a single eight bit wide tristate buffer. This means that a single input port can read the latches of both interrupts. Similarly a single output port can clear the latches of both interrupts. This can offer a considerable saving in the components required. The controller was designed to generate interrupts at level three and four.

The vectored interrupt controller of the second design was built and tested. The design was later abandoned because it is not possible to build two controllers on the same board that contains the two ported memories. The simpler autovector design was favoured because of the fewer components that it used.

3.10 Input and Output control ports

The processing nodes must be given a unique identifier. This enables identical software to be used on all the processors. The software can differentiate the identity of the host processor by reading an input port which is hardwired with a unique code. Three bits

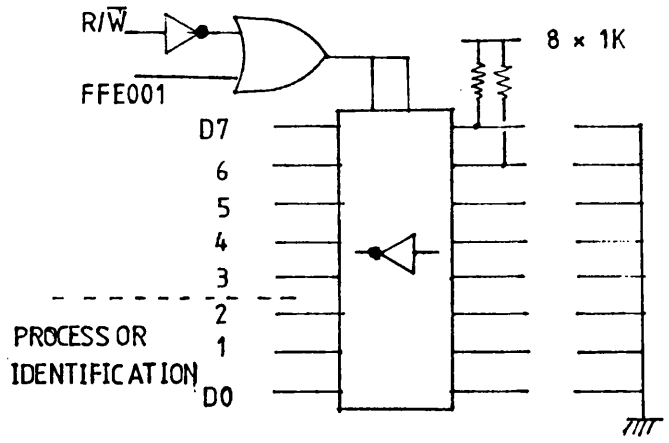


FIG. 3.17 INPUT PORT

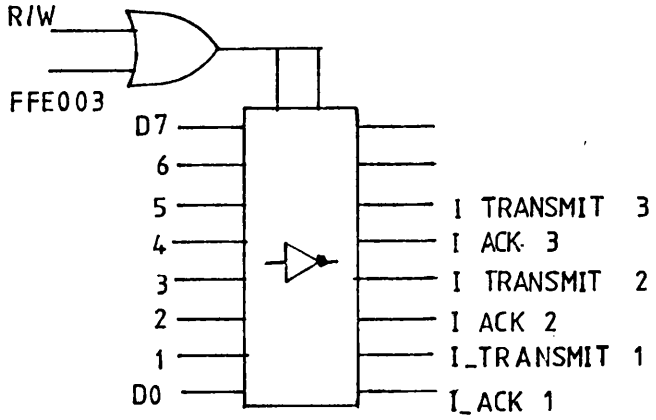


FIG. 3.18 OUTPUT PORT

are needed to give a number of one through six. The remaining bits can be used for other purposes. The circuit consist of a single 74LS245 tristate buffer(fig 3.17).

The function of the interrupt controller is to process the interrupt request. Some means of signalling an interrupt to the neighbouring processor is necessary. The simplest way is to use an output port. The port is required to generate a negative going pulse of ample duration. A single output port is sufficient to generate the six interrupt request signals. These signals are routed to the destination processing nodes through the appropriate cable ports. A 74LS245 tristate buffer is used (fig. 3.18).

3.11 Address decoder

The address decoder has to generate the following address select signals:

1. memory 1 $FFE800-$FFFFFF
2. memory 2 $FFF000-$FFF7FF
3. memory 3 $FFE800-$FFEFFF
4. interrupt controller level5 $FFE005
5. interrupt controller level4 $FFE007
6. interrupt signal port $FFE002/3
7. input/output port $FFE000/1

Figure 3.19 shows the circuit of the decoder.

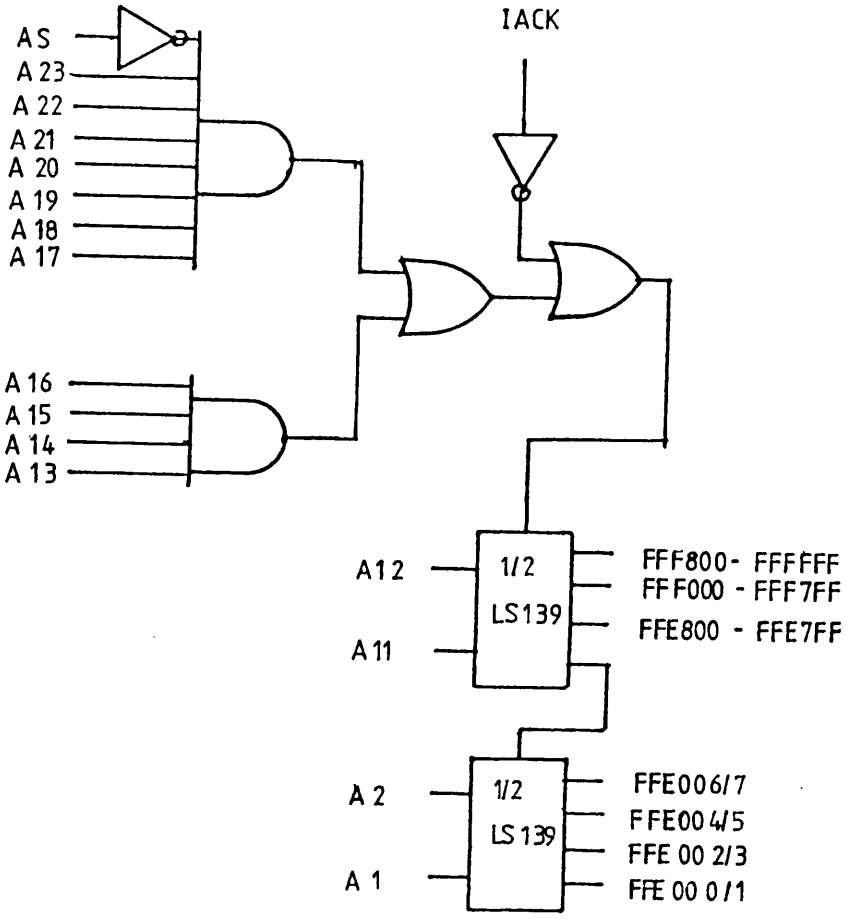


FIG. 3.18 ADDRESS DECODER

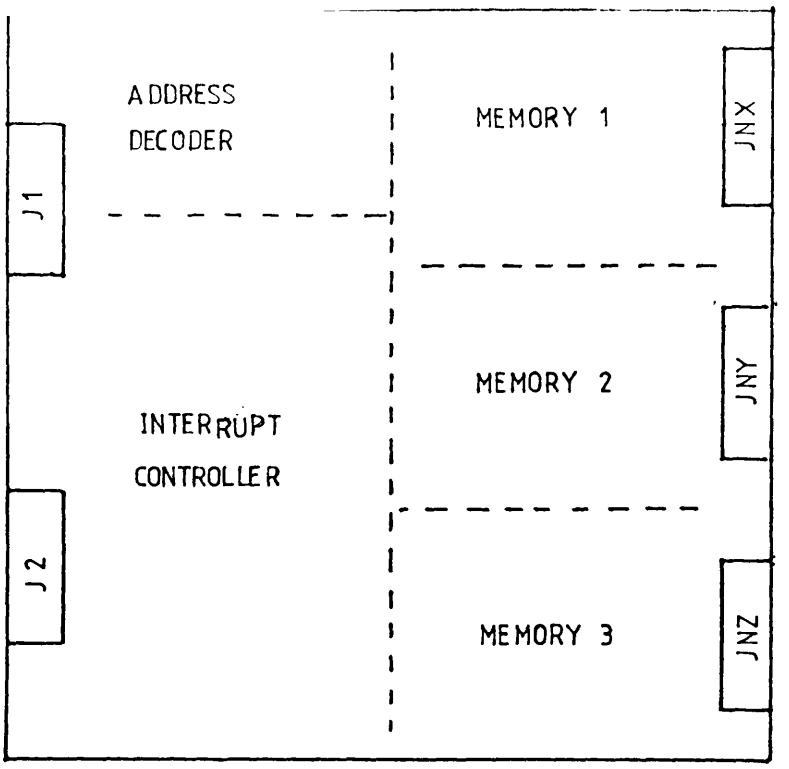


FIG. 3.20 MEMORY BOARD

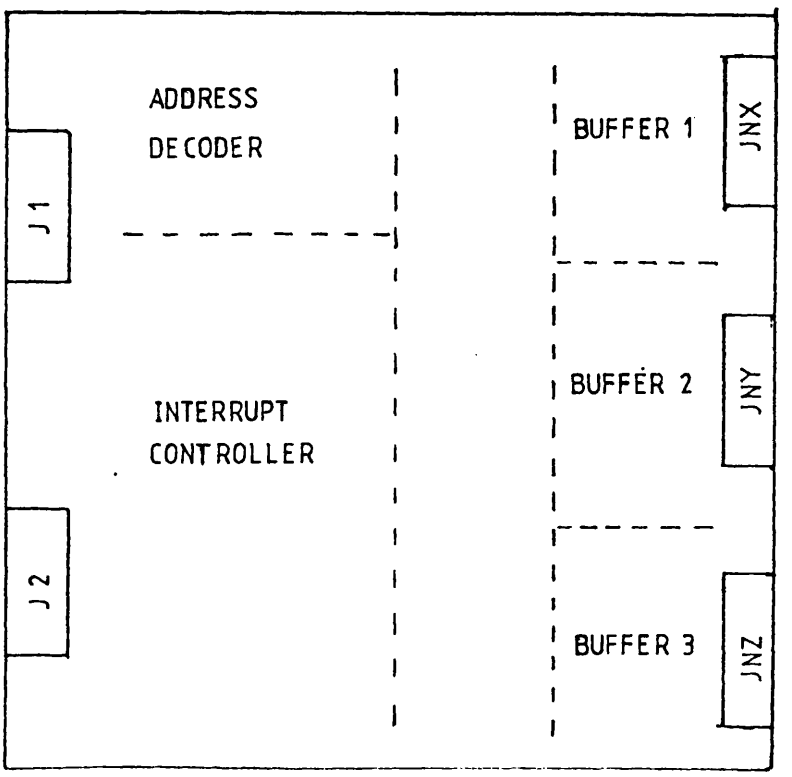


FIG. 3.21 BUFFER BOARD

3.12 Summary

The implementation of a set of communication interface has been achieved, thus satisfying the hardware requirements of the multiprocessor system. The aim of building an efficient communication system at a low cost has also being met. Below is the summary of the design.

The communication interface is made up of six circuit boards of two types, three of each type of boards. The first type contain the shared memory circuit and the other contain the buffers that interface the remote shared memory to the local processor bus. There are additional functions common to both type of boards. The functions are-

1. Two interrupt controllers;
2. parallel input/output ports for control purpose.

Figures 3.20 and 3.21 show the functional layout of the boards.

CHAPTER 4 SOFTWARE

4.1 Introduction

A software kernel is required to drive the multiprocessor. The functions of the kernel are to coordinate the interprocessor communication and to allocate processes to processors. The kernel must be able to support the generation of parallelism by the application program. Suitable interfaces provide the link between the application program and the kernel. Chapter one describes the various ways of exploiting parallelism. Of these, only a few are suitable for the multiprocessor network that is being investigated.

The method that seems to be most suitable for the multiprocessor network is demand driven computation. The parallelism is generated using the divide and conquer method. The divide and conquer method has the capability to generate an enormous number of processes which could easily exceed the number of available processors. The kernel must be able to allocate dynamically processes to processors. The scope of a process is from the moment it is activated to the moment it is temporarily suspended and from the moment it is reactivated until the moment it is terminated. During its active state, the process can be run uninterrupted. Even though there can be more processes than processors it is not necessary to run all the processes in parallel by multitasking. At the expense of losing some parallelism, the processes that are capable of being actived can be made to wait in a

queue. A scheduler then schedules the execution of the process whenever a processor becomes available.

The kernel structure is very dependent on the method of exploiting the parallelism. It is therefore appropriate to start the description of the design and implementation of the kernel from the high level end. This requires defining a hypothetical high level language that is capable of describing the parallelism generating process. The next step is to define a virtual machine that support the language. Finally the actual structure of the kernel can be defined and coded. The codes for the communication routines must be able to exploit fully the available hardware.

4.2 Language

This hypothetical language provides facilities for automatically extracting parallellism inherent in an application program. There are languages that exploit the architecture of the machine and parallel evaluation strategy of the problem to be solved. For example Val (27) for the data flow architecture and Flow graph Lisp (2) for applicative architecture. The main source of parallelism that is going to be investigated on the network is recursive subdivision. The hypothetical language can be based on the syntax of Lisp. Darlington and Reeve (1) described parallel reduction using a first order recursion language loosely based on NPL (28). The same approach can be taken here. However the

hypothetical language to be described will be based on Pascal. The reason for this is that investigation of the network will be more inclined to numerical computation than to symbolic computation.

Consider a PASCAL program of figure 4.1. The function T represents a processing task. By not allowing global references or assignments to be made from within the function body, several instances of the function can be created. If there is more than one function invoked simultaneously, the functions can be executed in parallel. Inside the body of T, a PARBEGIN...PAREND construct allows simultaneous recursive calls on T. This is the only facility provided for invoking parallel execution.

Function T behaves no differently from a normal function. It expects an argument when called and returns a result on completion. Since the function T represents a processing task, the passing of argument and result actually represent intertask communication. Intertask communication can only occur between a parent task and its children tasks. A child task can reside on the same physical processor as, or on a neighbouring processor to, the parent task. At the language level there is no distinction between the two.

```
PROGRAM Tree
VAR x:.... ;

FUNCTION T(.... ):.... ;
  VAR .... ;
  FUNCTION .....;
  PROCEDURE .....;

  BEGIN
  ...
   PARBEGIN
    a := T(    );
    b := T(    );
   PAREND
  ...
  END;

BEGIN
  ...
  X := T(.. ) ;
  ...
END .
```

Figure 4.1

The argument or result could either be simple data or complex structures such as arrays. With reference to the divide and conquer algorithm, the size of the

argument or result will decrease with each call. A dynamic array facility will optimise the use of storage.

A parallel function can return a whole array or a section of an array. To simplify the assignment of arrays the following statements are provided.

```
A[i:n] := function T(   )
A[i:n] := B[j:n]
M1[i:n,j:m] := M2[p:n,q:m]
```

where i, j, p and q are the first elements in the array and,

n and m are the number of elements to be transferred.

Declaration of local functions and procedures is allowed in the function T. These functions and procedures serve as utility routines.

4.3 Kernel

In order to support the hypothetical language described, a suitable kernel must be built on top of the hardware. The basic responsibilities of the kernel are

1. management of dynamic tasks,
2. interprocessor communication.

A task is created in response to a demand for a computation. The task is suspended when it spawns subtasks. The original task will remain in this state

until it has received results from its subtasks. The task proceeds until it reaches the end of the computation. Some basic mechanisms are required for supporting the computation. First is a system of task descriptors that hold information about tasks. This would be equivalent to a data stack for Pascal (32). In a way invoking the task is similar to calling a procedure or function. However it is not as straight forward as executing a call instruction in a processor. A task invokes subtasks by creating instruction packets. At the termination of a task, a result packet is issued to the parent task. If the packet source and packet destination are different processors, the kernel will route the transfer through the appropriate communication path.

4.3.1 Components of the kernel

The kernel can be broken into three major components. They are the scheduler, sender and receiver processes. Logically they are parallel processes in relation to each other. However it does not mean that three processors are required to realise a processing node, nor is a multitasking executive required to emulate the three processes. The scheduler resides in the normal processor state, the sender and receiver are interrupt processes.

Data structures are required to maintain the task descriptors and communication packets. The data

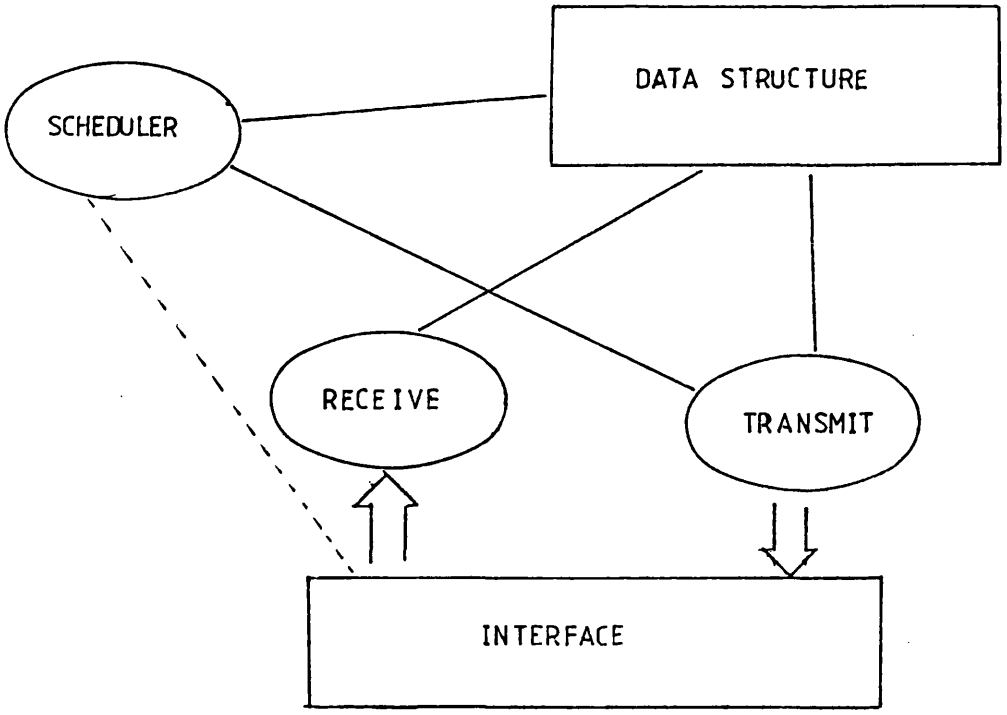


FIG. 4.2 KERNEL PROCESSES

structures are accessible by all the three processes. Figure 4.2 shows the relationship of the three processes with the data structures and the communication interface. The scheduler has a direct path to the sender through which the scheduler pass the information for transmission. There is no direct path between the receiver and the scheduler. The information received by the sender directly updates the data structure. The scheduler has a direct link to the communication interface denoted by the dotted line in the figure 4.2. This link enables the scheduler to read and write directly into the shared memory for control purposes.

4.4 Data structures

4.4.1 Task descriptor

The first of the data structures is the task descriptor. The purpose of the task descriptor is to maintain housekeeping information as well as the variables used by a task. This information must be held valid from the moment the task is invoked until the task is killed. The problem associated with this is in organising the store that will contain the task descriptors. In sequential evaluation, the chronological order in which the functions are invoked enables the data frame for the functions to be held on a stack. But here the order in which the tasks are invoked is less well defined. The prospect of a subtask

migrating to another processor made the problem more difficult. A lavish solution is to allocate a task descriptor for every task that would be created and not reuse the space left by an inactive task. Clearly this is not a feasible solution. Garbage collection can be used if there is not enough space. However if the task descriptor for every task is constant in size, the space left by an inactive task can be reused. The task store consists of a linked list of free task descriptors. A task descriptor is taken off this list whenever required. The task descriptor is returned by rechaining the descriptor onto the list. This occurs whenever the life of a task related to the descriptor has ended. This method of storage management will always take a constant time to recover a used descriptor cell. The guaranteed response time of this storage management method is favourable to garbage collection because uneven response time can effect the way the task are distributed. In the high level language abstraction described previously, a task function is allowed to contain local procedures and functions. Since the size of the data space allocated is fixed, local recursion is not possible. Handling of dynamic array structure will be treated later.

A task descriptor is definable by a unique address. The address specifies the host processor and an index relative to the base of the task descriptor store. This unique address is used by the communication packet to

specify the source and destination.

The fields inside a task descriptor are as follows-

1. Next pointer - forms the chain to the next descriptor cell.

2. Task descriptor index - indicates the descriptor index of the cell. If the descriptor is specified by its absolute address, this field provide a quick way of determining the descriptor index.

3. Parent node, parent index and task number - these fields form the complete address of the parent task. Parent node and parent index is the address of the parent task descriptor. The task number specifies which of the subtask from the parent is the current task.

4. Subtask count - indicates the number of subtasks that are created by the current task.

5. Entry pointer - this fields holds the absolute address of the code of the task to be executed. On first being created this pointer contains the address of the beginning of the task. On reactivation, it contains the address of the reactivation point.

6. Variables - this holds all the variables for the computation. These includes the arguments, receptacles for results from subtasks and local data.

Special treatment is necessary in organising the local variables. Since the size of the decriptor cell is fixed, the space for dynamic data structures must be allocated elsewhere in a heapspace. Reference to the dynamic data structure is by a pointer. Some means must be provided to differentiate between an absolute value and a reference. This differentiation is not necessary for an application program. Assuming a compiler is available for the hypothetical language, this differentiation would have been done at compile time. However the communication routine requires further information in order for it to transfer the data correctly. Every data item must therefore carry additional information specifying whether the data is an absolute value or an array. For the array, it also specifies the size and dimension. The data field contains the pointer to the heapspace.

4.4.2 Instruction and result packet

The instruction acts as the mechanism for invoking a task. A task that wishes to generate subtasks does so by creating instruction packets. The instruction is a record with the following fields-

1. parent node, parent index and task number
2. argument (number of data, data1, data2, ..dataN)

The first set of fields specifies the source of the

instruction. This is used by the child task to identify the destination of the result. The arguments consist of more than one data item. The data can be simple variable or complex ones. It is also necessary to make the size of an instruction packet cell constant. A sufficient upper limit of the amount of data can be arbitrarily fixed, but the variable size of the dynamic data cannot be accomodated. Therefore the data are passed in the instruction packet in a similar form to that in the descriptor. The instructions can either be held on a queue or a stack. Instructions held on a queue results in a breadth first evaluation. Holding instructions on the stack produces the following effect. If the instructions are executed on a local processor, the evaluation will be depth first, but if the instruction is executed on a remote processor the evaluation will be breadth first in relation to the other instruction created simultaneosly.

The result packet consists of the destination and a single data item. As the result already specifies the destination, the result is sent immediately it is produced.

4.5 Scheduler process

The primary role of the scheduler is to retrieve instructions from the stack and run the task created by the instructions. Before the task can be executed, the task descriptor has to be set up. Instructions can be

obtained locally or from one of the three neighbouring nodes. A scheduler from one processor cannot directly access the instruction stack of another processor. In order to access an instruction from another processor, the scheduler of the requesting processor makes a request to the processor concerned through the communication link. The job of servicing a non local request is also handled by the scheduler. Only an instruction which has not been made into a runnable task can be transported, because a runnable task is allocated a task descriptor locally. In order to differentiate between the instructions that can be transported and the instructions that cannot be transported a seperate list is required. An instruction that has been made runnable is placed on a queue. A runnable task can also be created by a task that is reactivated after suspension. The scheduler inspects the runnable task queue after first looking at the instruction stack. The actions of the scheduler will now be described. The first phase of the scheduler is to retrieve instructions. Initially the local instruction stack is inspected. If an instruction is available a task descriptor is allocated, and the information carried by the instruction is copied into the task descriptor. The source of the instruction is local. In the case of the argument specifying a dynamic array, there is no necessity for generating new space for the data. The pointer carried inside the

instruction packet is valid. The index of the task descriptor is placed on the runnable task queue. If there are no instructions available, requests are made to the neighbours. The request is made by issuing a communication packet carrying the appropriate information. A normal communication transaction can take a significant processing time in both processors. Since the two processors are linked by shared memory, a flag can be reserved inside the memory to indicate an instruction request. A requesting processor will set this flag to signal that it is requesting an instruction. However the response to this request is not instantaneous because the scheduler of the receiving processor must have arrived at the appropriate phase before this request can be serviced. The empty processor can go through several iterations of the scheduler loop before it is granted an instruction. In order to restrict access to the shared memory, the scheduler can only set the flag once before the request is granted. This is done by maintaining a separate set of flags in the main memory. A processor receives an instruction from a neighbour not through the scheduler but through the receiving process.

After the scheduler has obtained an instruction from its local stack and generated a runnable task it now attempts to distribute the remaining instructions on the stack to its neighbours. This is the complementary action in response to the request made by the

neighbour. The scheduler inspects the request flags in all the shared memories in turn. Upon recognising a request and identifying the neighbour which made the request, the scheduler fetches an instruction from the stack. From the instruction, a transmission packet is made. The packet is passed to the sender process for transmission to the neighbour processor concerned. This action is carried out for every neighbour. However if after transferring one or two instructions the stack becomes empty the action is stopped and moves on to the next phase. The response of the receiving neighbouring processor to the transmission of an instruction is to go into the receiving process. The receiving process first identifies the nature of the packet. In receiving the instruction, the scheduler of the receiving processor is not involved. The receiving process allocates and sets up a task descriptor for the instruction. In addition storage space is allocated for dynamic array structures that can be contained in the instruction. After the set up, the index of the task descriptor is mounted on the runnable task list.

The last phase of an iteration of the scheduler loop is to retrieve a runnable task. The runnable task list contains the index to active task descriptors. The task is entered by jumping to an address specified in the task descriptor. During the execution of a task, the action of the scheduler is thereby suspended. The scheduler is reentered when the task is suspended or

terminated. However the sender and receiver processes can coexist with the task because they are interrupt processes. If the runnable task list is empty the whole phase of the scheduler is reexecuted from the beginning. The scheduler process is described in Pascal notation below(fig. 4.3)

```
WHILE true DO
 BEGIN
  IF active task queue is empty THEN
   BEGIN
    IF instruction queue is not empty THEN
      BEGIN
       get instruction from local queue
       IF intruction queue still not empty
       THEN
        try transfer instruction to neighbour
      END
    ELSE
     request instruction from neighbour
   BEGIN
    get runnable task and execute
   END
 END
```

Figure 4.3

4.6 Task processing

When the task is activated all the data structures

associated with it have already been set up. The size of the problem determines whether splitting the problem is possible. From within the task, a facility for spawning subtasks is provided by a system procedure. It is important that the normal processing state is protected during the spawning process. For example a task can spawn eight tasks in parallel. There is a counter in the task descriptor which is initiallised to the number of subtasks generated. Every time a result from a subtask is obtained the counter is decremented. When the count reaches zero the parent task is reactivated. If the spawning process is not protected, there will be a possibility that the parent task is reactivated prematurely. For example the first task spawned is grabbed by a neighbour. If the outcome of subtask 1 is returned before further subtasks can be created, the parent task is immediately reactivated inadvertently.

A task that cannot be split or a task that is reactivated will eventually reach a point that requires them to return result to the parent task. The parent can be local or on a neighbouring processor. In the case of a local parent, the outcome of the result can be notified by directly accessing the parent task descriptor. The result is passed to a neighbouring processor using the communication processes.

There are several more areas where the normal processing state should be protected to safeguard the

integrity of the data shared by the normal processing state and the communication processes state. The task descriptor can be both allocated by the scheduler and the receiving process. Therefore the allocation of task descriptors by the scheduler must be protected.

4.7 Communication

The memory window provides a two way communication path between two processors. The incoming and outgoing paths are logically separated. There will always be the possibility that both processors attempt to send at the same time. Since there are separate paths provided, there will not be problems in gaining access to the channel. A successful communication would require the co-operation of both the talker and listener. If both parties talk to each other simultaneosly, even though not on the same channel, the communication would still fail. The analogy to this argument is that of a telephone conversation.

A scheme of organising communication in the shared memory is sought. It is helpful if it can be proven that the scheme will work. Proving correctness of parallel program is still at an early stage of development (26) (29). For this reason although a formal proof is not given, an attempt is made to deduce that the scheme will work. This is done by basing on an analogy of a more primitive mode of communication.

Let us view the communication to be between two

parties using pneumatic tube normally found in department stores. The information that is being communicated are messages written on pieces of paper and the communication is bidirectional. Two pipes are provided, one for each direction of transfer. Both parties can send messages simultaneously without any problem. One restriction is imposed on the use of the pipes. No further message can be sent down the pipe unless the receipt of the previous message has been acknowledged. Some means of signalling the conditions of the pipe is therefore necessary. The conditions are -

1. message acknowledge.
2. message available.

'Message acknowledged' would mean that the outgoing pipe is free for further sending of messages. The signal for 'message acknowleged' is transmitted at the instant the receiver takes out the message. It is not necessary that the sender is forever wanting to send messages continuosly. It may retrieve the message at a later time but not necessarily immediately. Therefore it is sufficient for the 'message acknowledged' signal to set a flag at the sender's end.

The arrival of a message at the receiver generates a message available signal. In order not to block the pipes, the message must be removed immediately. The operator at the receiver's end should preferably be interrupted rather than performing an inspection whenever he or she is free. The later is equivalent to

polling.

Let us go back to the original problem of simultaneous transmission of messages. The operators at both ends pop the drums that contain the messages into the pipes and release a burst of compressed air. At this instant neither realises that they would be expecting messages from each other. Thus, they return to their normal duties. However, a moment later they are interrupted by a ring on the bell signalling the arrival of a message in their incoming pipes. Retreiving the message is given a high priority, knowing that it could block further incoming messages.

From the discussion above, it is clear that both parties are still able to send to each simultaneously. The scheme will also work if there are several messages to be sent one after the other, if the messages are queued. Multiple transmission will now be illustrated. Continuing from the point where the messages were retrieved, the operators observed that their respective message acknowledge flag is set. This signals that further transmission can be performed. The next messages on the queue is fetched and the sending procedure is repeated.

The scheme can be applied to the shared memory communication. The two logical channels in the shared memory were as illustrated by the pneumatic pipes above. In the discussion presented, it can be deduced that the scheme is secure and free from deadlock. The

signalling can be realised using interrupts. Two levels of interrupt are required, one level for 'message acknowledge' and another level for 'message available'. Sending has higher priority than receiving, so the 'message acknowledge' interrupt is placed at the higher level.

One aspect which has not been illustrated is the necessity for the sending process to be uninterruptable during operation. 'Message received' interrupts from the other channels cause no problem because they are blocked by the hardware. The 'message acknowledge' interrupt has a higher priority level so as not to allow 'message available' interrupt to cut in during a sending operation in order to safeguard critical data region.

A method of describing communication at a higher level is the rendezvous concept. Rendezvous stipulates that both the sender and receiver must express their will to communicate. The task processing operation in addition to the communication involved can be described using rendezvous. For the sake of discussion, let us start with one processor which has just created subtasks and its neighbours are trying to grab these tasks. Prior to this instant the idle neighbours had already expressed that they require instructions. Assume this is expressed by a high level statement - RECEIVE(instruction). No further activities can be carried out unless there is a corresponding

SEND(instruction) executed on the sending end of the channel. After spawning its subtasks, the task is suspended by executing a RECEIVE(result). The processors which execute the susbtasks return the results by - SEND(result).

The idea of the whole exercise is to keep the physical processor as busy as possible. Logically the state of the task may indicate a wait, but to make effective use of the available processor power, the physical processor must be redeployed for other tasks and yet able to resume the logical wait.

4.8 Dynamic storage management

The task descriptor primary function is the storage of housekeeping information and local variables associated with a task. The housekeeping data and simple local variables occupy a fixed storage size. However the storage required for the dynamic array structure cannot be determined at program start up time. There is a strong argument for keeping the task descriptor size constant. Although the task descriptor has a similar function as a stack frame in a P-machine, the behaviour of the task descriptor with time may not be easily predicted. The order in which task descriptors are deleted may not have a simple relationship with the sequence in which they are

created. This is unlike the stack frame in a P-machine where the movement of the stack is well defined chronologically. The idea of making the task descriptors of equal size cells makes it easier to manage. Initially the task descriptors are linked as a large continous chain. A request for a task descriptor retrieve the front most cell from the chain. Returning the task descriptor is performed by simply putting the cell back into the chain. Since the task descriptor is of constant size the space for the dynamic array must be placed elsewhere. A heap space is allocated which is common to all the tasks for the dynamic array. Associated with the heap space is a memory allocation list. Initially this list contains one entry which describes one large heap. The entry consists of the pointer to the first position of the heap and its size. When a request is made the portion of the heap of the size required is extracted. The entry in the list now indicates the balance. Assume that after a few request is made the first allocation is to be returned. The policy employed is to chain back the memory. There now exist a gap between the end of the returned memory and the remaining heap. The pointer and size of the returned memory is to be put on the list. The list is arranged such that the entry for the lower memory preceeds the entry for higher memory. With the next allocation returned the approppriate position in the list is first determined. Then a test is made to see

whether the preceeding entry is continous with the memory to be returned. Similarly the entry for the next higher location is tested. If the test succeeds, no new entry is entered but the existing entry is amended to indicate a newly formed block. The next allocation of storage will attempt to find the first returned block that fix the size requested or a larger block with the least difference.

The danger that can occur with dynamic storage management is storage fragmentation. There is no danger of fragmentation if the size allocated is constant or in multiple of some fixed size. The size of memory allocation list must be large enough to cope with any demand. The possibility of overflow is reduced if the way the memory is returned always attempt to rejoin returned blocks.

4.9 Program development

To test out the ideas developed so far two problems were chosen, quicksort and matrix multiplication. To carry out the test, the kernel was first defined. Each of the problems was then built on top of the kernel. All programming was done using MC68000 assembler. Program development was carried out under Tripos operating system running on one of the processors. Disc facility for the processor running Tripos was provided by an Ithaca S100 system. The S100 system runs a file server which can serve more than one processor

simultaneously. The file server was written in colaboration with Dr. Jed Marti (34). Two MC68000 processors are linked to the Ithaca by parallel ports. The parallel ports were designed and built in colaboration with Dr. D. Milford (22). The other MC68000 processors can be linked to the Ithaca by RS232 lines. Dedicated disc system was also provided on one of the MC68000 processors. The disc system consist of two eight inches drive.

There are two reasons for implementing the test programs in assembly language. The first is the need to have maximum control on the hardware. Secondly the program must be stand alone and do not require the assistance of the operating system. The program need not be totally in assembly language for it to have maximum control on the hardware. The program could have been written in BCPL and the hardware sensitive routine coded in assembly language. But this requires that all the processors must be running some limited form of the operating system. In principle all the six processors can be linked to the file server but with that kind of load the response of the file server is excruciatingly slow.

The program is initially developed with two processors running Tripos. Both machines are required to run Tripos because of the need to use the interactive debugger. This was done with the assumption that if the program works on two machines, it should

work on N machines. The set up for bringing up more than one machine consists of one processor running Tripos which allow the loading of object code from files. The other processors contain a loader in rom which loads the object code through the communication interface.

A facility is provided to synchronise all the processors at start up time. Recalling from the hardware section every processor is provided with an input port. A spare bit of the input port is used for this purpose. The main processor has an extra bit output and this is wired to the 'sync' input of the other processors.

The implementation of the kernel and the test programs was not a one pass process. From the initial coding of the program to having a minimal two processor system running took several cycles of debugging and re-coding. In principle, a working two processor system would have exercised a high percentage of the program codes. A debugged two processor system should run for system with more processors. In reality this was not the case. The sections of program that handle task distribution and communication were not totally tested. Finding the problem codes when more than two processors are involved are extremely difficult with the available debugging facilities. The initial objective of the experiment was to test quicksort and parallel matrix multiplication program on a six processors system.

Unfortunately this was not achieved due to the untimely failure of the hard-disc system. Although the file server system was partially restored to floppy disc, the speed and limited file size allowed on the floppy disc restricted the productivity of program development tremendously. In the experimental section to be described forth are the results obtained with fewer than six processor configurations.

4.10 Experiments

4.10.1 Quicksort

Below is the description of the quicksort using the hypothetical language of section 4.2.

```
Program Quicksort;
 const n=    ;
 type index=1..n ;
      item =record of
              key,value:integer
             end ;
 var a:darray [1..n] of item ;

 function sort(l,r:index;a:darray [1..n] of item) ;
  var i,j:index; x,w:item;

  begin
    i:=l; j:=r;
    x:=a[(l+r) div 2];
```

```
      repeat
        while a[i].key<x.key do i:=i+1;
        while x.key<a[j].key do j:=j-1;
        if i<=j then
          begin  w:=a[i]; a[i]:=a[j]; a[j]:=w; i:=i+1;
                 j:=j-1;
          end
      until i>j ;
      if l<j then a[l:j-l]:=sort(l,j  );
      if i<r then a[i:r-i]:=sort(i,r  );
    end ;
  begin
    a[1:n]:=sort(1,n)
  end.
```

Figure 4.4

The quicksort program was run on one and on two processors for various sizes of unsorted arrays. The unsorted arrays were generated randomly.

The followings are various time taken to sort the arrays on one and two processor configurations.

1. One processor

Problem size	50	100	150	200
Time*20 msec	4	7	11	15
Task executed	45	89	132	174

2. Two processors

Problem size	50	100	150	200
Time*20 msec	3(3)	5(5)	7(6)	9(9)
Task executed	24(21)	40(49)	63(69)	92(82)
Task xferred	3(4)	11(2)	2(3)	1(11)
Speed up	1.33	1.4	1.6	1.7

note: figures in brackets are values for second machine.

The result on quicksort shows that the efficiency increases with larger problem size. A simple explanation of this is that the larger problems can sustain longer parallel computation. The initial splitting may not generate tasks of equal size. The distribution of tasks is very dependent on the quicksort problem itself as opposed to the effect of the network. The processor that has the smaller task would sustain shorter parallel computation than the processor that has the bigger task. In order to proceed with further work, the now idle processor has to request a task from the other processor. The chance of getting a computationally small task is high. Therefore there will be heavy communication between the two processors in order to keep both processors busy. It is expected that the communication would reduce the overall efficiency. However this was not evidenced from the result obtained. With problem size of 100 and 200,

the results show that one processor received a fairly large number of tasks from the other processor. There is little task movement with problem size of 50 and 150. However the efficiency at problem size of 100 is higher than the efficiency at problem size 50. Similarly the efficiency at problem size of 200 is higher than the efficiency at problem size 150. A possible explanation to this is that the overhead associated with communication is minimal when compared to the computation of the problem.

4.10.2 Parallel matrix computation

The divide and conquer method can be applied to matrix multiplication. Consider the multiplication of 2 by 2 matrices. The multiplication is definedas follows-

$$\begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \times \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

where

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

Recursive subdivision can be applied on larger size matrices if the size N satisfies $N=2^m$ where m=(1,2,3,..). The multiplier and multiplicand matrices are each divided into four quadrant where the quadrants represent A_{11} ,A_{12} ,... . At each level of recursion there will be eight subtasks generated. The

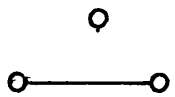
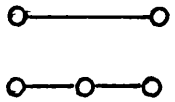
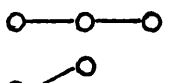
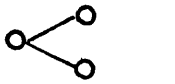
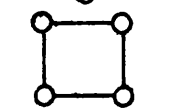
recursion will terminate when the size of the matrix is 2 by 2.

```
Function pmult(n:integer;A,B:mat):mat;
 Var C:mat;
 Begin
   If n=2 Then
     Begin
      C[1,1]=A[1,1]*B[1,1]+A[2,1]*B[2,1];
      C[1,2]=A[1,1]*B[1,2]+A[1,2]*B[2,2];
      C[2,1]=A[2,1]*B[1,1]+A[2,2]*B[2,1];
      C[2,2]=A[2,1]*B[1,2]+A[2,2]*B[2,2];
      pmult:=C[1:2,1:2]
     End
   Else
    Begin
     m:=n/2
     C[1:m,1:m]:=pmult(m,A[1:m,1:m],B[1:m,1:m])
                +pmult(m,A[m+1:m,1:m],B[m+1:m,1:m]);
     C[1:m,m+1:m]:=pmult(m,A[1:m,1:m],B[1:m,m+1:m])
                +pmult(m,A[1:m,m+1:m];B[m+1:m,1:m]);
     C[m+1:m,1:m]:=pmult(m,A[m+1:m,1:m],B[1:m,1:m])
               +pmult(m,A[m+1:m,m+1:m],B[m+1:m,1:m]);
     C[m+1:m,m+1:m]:=pmult(m,A[m+1:m,1:m],
                B[1:m,m+1:m])+pmult(m,A[m+1:m],
                B[m+1:m]);
     pmult:=C[1,n:1,n]
    End
End;
```

Figure 4.5

The parallel matrix multiplication problem defined in the hypothetical language in section 4.2 is shown in figure 4.5.

The following are results obtained for a 16 by 16 matrix multiplication.

configuration		time*20 msec	speed up
1		76	1
2		50	1.56
3		44	1.73
4		37	2.05
5		36	2.11

The general trend of the result is that the speed of the machine increases with more processors. However a different configuration for the same number of processors produces a different speed up. For the three processors system, the straight line configuration produces poorer speed up than the binary tree configuration. This shows that the straight line configuration cannot distribute tasks efficiently compared to the binary tree configuration. The initial splitting of a problem generates the highest potential for work. In the straight line configuration the rightmost processor will never have the chance to grab the task that was generated with the initial splitting. The tasks that reached the rightmost processor are computationally small. In order to sustain further computation there will be heavy communication involved

with the middle processor. The frequent communication reduces the efficiency of the machine. For a similar reason, the four processor system in a square configuration does not show great improvement over the three processor binary tree configuration. The processor diagonal to the processor where the task is injected, obtained tasks that resulted from at least three subdivisions.

4.11 Conclusion

Consider the case of the two processor configuration for both quicksort and matrix multiplication experiments. It would be expected that the machine performance for quicksort is less than that for the multiplication problem due to the sparse evaluation tree for the quicksort. But on average the performance for both problems is similar. In a two processor system, there should be enough parallelism in both problems to saturate the processors. In a larger system it can be safely assumed that the matrix multiplication problem would produce better performance because the regular expression tree could sustain enough parallelism on all the processors.

CHAPTER 5 DISCUSSION

5.1 Introduction

A survey (5) (12) (15) (21) (56) of the work done on the development of multiprocessor and parallel computers can be loosely categorised into two branches. The first is developing machines for a specific applications. Examples of these are dedicated networks of multi-microprocessors modelling a specific problem and the vector and array processors for number crunching applications. The other branch is developing machine for general purpose applications based on new computing models. In both branches of development, the hardware configuration derived has a direct relationship with the problem to be solved or the computing model. The starting point for the research carried out in this thesis is a multiprocessor configuration proposed by Bowyer et al for some applications in graphics. The realisation of the processor configuration did not require a major conceptual development exercise. The nature of the problem encountered was more of practical difficulties. The next step in the exercise was the design of the software for driving the multiprocessor. The concept in the design of the software was not confined to graphics applications only but to a much wider scope of general purpose application. The development of the software system was more involved with concepts and the theoretical aspects of computing model. On the implementation side of the software system, practical

problems were encountered in program development and debugging the multiprocessor. This research would belong to the first category mentioned above since the hardware derived was for a specific application. However, the requirement for suitable software for driving the multiprocessor necessitated venturing into the second category of development mentioned.

To date what has been achieved in this research is the construction of a multiprocessor hardware within a small budget and the development of the software required to run the system. The state of the software developed is ample to test the multiprocessor and run experiments for the purpose of evaluating the performance of the system. Subsequent text in this chapter presents a discussions of the degree of success of the Bath system as a multiprocessor and the further development possible on the machine.

5.2 Performance

The experiments done on quicksort and matrix multiplication show that the machine does gain in processing speed over a single processor(section 4.10). The only form of test is to measure the time the multiprocessor to complete a problem. Since the system can be set up with a single machine, time measured for various processor configurations can be compared with the time for a single machine. By no means is the result of the experiment conclusive. A more

comprehensive set of results can be obtained if the traces of activities of all the processors were recorded. This form of test will be able indicate any bottlenecks in the system. To perform this test require a more elaborate hardware set up. Section 5.3 described a proposal for such a hardware configuration.

5.2.1 Effect of interface hardware on performance

The primary measure of the efficiency of a multiprocessor must be based on how much gain in speed is obtained over a single processor machine. The efficiency of the system is decided by the ratio of actual gain in speed over the ideal maximum possible. Ideally an N processor machine should be N times faster than a single processor machine.

There are several factors that decide the performance of a multiprocessor. The three main factors are-

1. The interprocessor link hardware must be highly efficient for communication costs to be kept low;
2. An ample amount of parallelism must be inherent in the problem to be solved in order to sustain parallel execution;
3. The software that is responsible for the management of tasks and communication must be efficient.

Our interprocessor link is provided by reasonably fast interface hardware based on shared memory. The interface is fast compared to serial or parallel input/output under program control. A normal memory read or write instruction takes four clock cycles. Since the interface introduce two wait states, one extra clock cycle is required to be added to the memory access timing. A MC68000 move memory to memory instruction for long data takes twenty clock cycles plus one extra clock cycle introduced by the interface. Below is an assembly language routine that is used to move a block of data from main memory to the interface.

```
loop    MOVE.L   (A0)+,(A1)+     21 clock cycles

        DBRA     D0,loop         12 clock cycles
```

The transfer rate for long word data that can be achieved by the routine above is calculated below-

Total clock cycles 33

Total time (8Mhz clock) 4.125 uS

The transfer rate is 1/4.125uS or 242.4 Kwords per second. The data transfer rate can be improved marginally by removing the wait state. A much faster data transfer rate can be achieved under direct memory access control. Assuming the memory cycle time is 200ns, a single word move takes 400ns (total read and write times). The transfer rate is thus 1/400ns or

2.5Mwords per second. This transfer rate is highly desirable. Even though the transfer rate under program control is one tenth of that under dma control, the speed is reasonably fast when compared to serial or parallel input/output lines. It can be concluded that the choice for the communication hardware does conform to the factor 1 described above, especially considering the low cost of the interface.

5.2.2 Effect of software on performance

It was mentioned in the introduction chapter that there are various ways in which parallelism can occur in a problem and the way the parallelism can be exploited. Divide and conquer is one method. The reasons for directing the investigation towards divide and conquer are-

1. The interprocessor configuration was conceived on the idea of divide and conquer computation;
2. The possibility of realising a general purpose parallel machine. It has been reported by several researchers (2) (4) (57) that a divide and conquer algorithm is capable of producing an exponential growth of parallelism in applicative program.

The software system developed is a kernel for a divide and conquer virtual machine. The kernel system seems capable of performing its logical function. Unavoidably there are overheads introduced by the system. The sources of overhead are the setting up of the task and

the communication between the processors. Incorporating a task scheduler in the kernel enables virtual task redeployment of physical processors but introduces further load on the machine. At the virtual level, the spawning of subtasks is similar to invoking a function in a sequential processor. However, in the multiprocessor an elaborate kernel is essential to manage the physical processor. Therefore a multiprocessor such as the Bath system will not reach the level of efficiency of a single processor. Better efficiency can be achieved in a multiprocessor where the mapping of tasks to processors is on a one to one basis. Such architectures are the binary tree processors (55) and systolic processors (52). In binary tree processors there is less overhead involved because there is no necessity for a scheduler. The parent processor start the children processors by implicitly sending the instruction and data to the children processors. The parent processor physically goes into a suspended state awaiting to be restarted by the children processors. Data structures such as the task descriptor is not required for maintaining the list of active and suspended task because the processor memory is exclusive to one task only.

It is highly desirable that all the component processors are evenly loaded and this is dependent on the task distribution mechanism. The distribution of tasks is handled dynamically by the system as follows.

The idle processors request tasks from the busy processors, rather than the busy ones pushing the tasks. The local processor always has the highest priority over locally generated tasks, but if there is more than one task available on the stack it is guaranteed that the scheduler will honour any request from the neighbours. If there are enough parallel tasks available, the scheme will ensure that all the processors are busy. The task distribution is accomplished by the processors mutually cooperating among themselves without the need of a control processor. This is obviously important for an asynchronous system.

The instructions are maintained on a stack. During a subdivision a number of instructions are generated and placed on the stack. All the instructions that are generated by a single subdivision process can be said to be contained within a subdivision frame. Parallel evaluation is guaranteed if some or all of the instructions within a frame are consumed before the next frame is created. The evaluation is breadth first. When there is no more demand from neighbouring processors the local processor can only consume one instruction each time from a frame before the next frame is created. This is in effect a depth first evaluation.

The cost of communication is dependent on the distance between the processors involved. The kernel

takes care of this by ensuring that the communication can only occur between adjacent processors.

5.2.3 Effect of interconnection topology on performance

From the experiments described in section 4.10, different processor configuration can affect the performance of the multiprocessor. It is not possible from the minimal result obtained in the experiments to extrapolate the result directly to the trivalent graph of maximal girth network. However an attempt will be made to analyse the network based from the experience gained from the simulation and the experiment on the actual hardware.

The idea of using the complex processor interconnection is to achieve even distribution of tasks among the processor. The maximum distribution of tasks will happen if there are enough outlets for the tasks to be dispersed. This condition is achieved if the number of ways the subdivision occur is less than the valency of the graph. In a girth g graph, all the processors will be loaded after g/2 levels of subdivision. Since all of the subtasks initially created are able to be dispersed, there will be none of the original subtasks remaining at the root processor. If the number of ways of subdivision is greater than the valency of the graph, the distribution of tasks will be less than ideal. Although the processors will

still be saturated after g/2 levels of subdivision for a girth g graph, there will be fairly large tasks remaining at the root processor. Maximum dispersion should ideally occur after the initial subdivision because the subtasks created are potentially capable of sustaining localised parallelism longest. The worst situation can occur if the depth of subdivision is less than g/2. The processors at a distant of greater than the depth of subdivision will never obtain a task because communication can only occur between processors of unit distance away. As an example, this situation can occur with a 16 by 16 matrix multiplication on a girth 10 network. There are only three levels of subdivision in the multiplication.

From the discussion presented above it can be seen that the trivalent graph network with maximal girth is not suitable for problems with the number of ways of subdivision greater than the valency of the graph. The analysis was based on a specific problem of matrix multiplication. In the case where the expression tree is less well defined as in the expression tree resulted from the execution of a reduction language (23) the task distribution is less predictable because the number of tasks created for every level of subdivision is not constant.

If the tasks are allowed to migrate more than once from the source processor, the dispersal of the tasks will not depend on the valency of the graph. However,

this method of task distribution has the disadvantage of requiring the use of intermediate processors for communication. It is important that the initial subtasks created are fully consumed. It is probable that the use of complex interprocessor topology may not offer very much benefit (41).

The discussion is by no means conclusive unless it is based on actual data obtained from experiments. The following section describes the necessary enhancement of the multiprocessor system in order to make further experiments possible.

5.3 System improvement

The Bath multiprocessor is essentially a test bed for exploring ideas on multiprocessors. However there are a few facilities both in hardware and software system that are lacking for it to be a suitable development system. The following subsections described the facilities that are desirable.

5.3.1 Hardware system enhancement

Further investigation needs to be carried out before a solid conclusion can be made regarding the efficiency of the interprocessor configuration and the multiprocessing kernel. As was previously mentioned, a trace of all the processors is useful. The way the tasks are distributed can be observed. In the experimental stage it is only feasible to build a

multiprocessor with a small number of processors. The only way the behaviour of a large scale multiprocessor can be observed is by simulation. However if a comprehensive set of data is obtained from small scale multiprocessor, the behaviour of a large scale system can be extrapolated. To be able to do more experiments and gather more data requires a better system set up. Figure 5.1 shows a possible hardware configuration. A control processor is linked to all the node processors. The control processor can interrupt all the node processors simultaneously and also perform a two way conversation with the processors. A simple serial link is ample to establish communication between the control and a node processor. The control processor regularly sends out an interrupt which suspends the processing on all the node processors. This interrupt should be on the highest level of interrupt used. The control processor can interrogate the node processors in turn. The states of the processors are recorded by the control processor.

5.3.2 Software development system

The software development system needs to be improved. Below is a proposal for improving the facilities on the multiprocessor system. Developing assembly language program is time consuming and laborious. Using high level languages(C, BCPL etc) which are normally used for writing operating system

and other system software should reduce the effort tremendously. The negative point of using such languages is that they require some form of operating system running on the processor. A suitable development system could be provided by a high level language cross development system which only requires minimum run time environment. This is more advantageous than a resident system. The cross development system can be hosted by the control processor of section 5.3.1. The choice of language is not critical nor is it necessary to modify the language to incorporate parallel constructs. However the high level language should have facilities for programming interrupt. The run time environment should be kept to a minimum to ensure low overhead but should incorporate some form of error reporting facilities. Using the proposed hardware set up mentioned, the occurence of error on any of the node processors should be reported to the control processor immediately. The control processor action would then be to stop all node processors and notify the console. From the traces of the node processors previously recorded, the programmer can ascertain the cause of the problem and suitable action can be taken.

5.4 General purpose programming

The high level abstraction described in chapter four is ample for the purpose of defining simple test programs. A more complete programming language for the

multiprocessor is necessary in order to investigate the behaviour of general purpose computing on the multiprocessor. The computational model incorporated in the kernel should be able to support an applicative style or reduction language. The kernel however may require some modifications.

Applicative programs exhibit some degree of natural concurrency. This concurrency is derived from multiple evaluation of function arguments and the behaviour of an applicative program on a multiprocessor is safe because there are no side effects. This form of concurrency can be observed in an expression f(a,b). When a and b are subexpressions, they can be evaluated in parallel. This form of concurrency of itself does not generate an enormous amount of parallelism. Divide and conquer is one method of deriving the desired amount of parallelism and has been decribed elsewhere in this thesis. Another method is through the appropriate use of data structuring (3). A function can have a sequence of arguments. An apply-to-all operator maps the function to all of the arguments in the sequence.

f>>( )( )( )

If the sequence is made up of a list of length n then there should be n tasks generated. Further parallelism can be generated if there are unevaluated subexpressions in the sequence.

The existing kernel of the Bath multiprocessor

incorporates a machine model that can support applicative style programming in a limited form. The present model supports divide and conquer evaluation on a single function. However it does not evaluate the arguments: the arguments are assume to be simple.

Task management relies on two mechanisms, the instruction and the task descriptor. The task descriptor represents a computational node. Execution of a program generates a tree of task descriptors. A task invokes subtasks by issuing instructions. An instruction contains the identity of the task that issues it and the data for the arguments. Since there is only one definition of function involved there is no necessity to have a seperate field for the name of the function. The number of arguments is fixed. However the data part of the arguments is variable in size. In order to simplify the management of the instruction stack the data part is separated from the rest of the instruction. The data part is maintained in a separate heap space. The task descriptor contains the task housekeeping information and also the local data for computation. For a similar reason, the data part of the dynamic variables in the task descriptor is maintained in the heap space

To allow for general purpose applicative language like Lisp the only requirement of the execution model is the capability to support both primitive functions and user defined functions. A more dynamic structure is

then necessary. The size of the instruction and the task descriptor are dependent on the function definition. The need to reduce the arguments requires an expression evaluator. The expression evaluator is called when a task is first started. The task can be suspended in the evaluator whenever there are subexpressions to be evaluated. The number of subtasks that can be invoked is dependent on the instruction. When all the subexpressions are evaluated the task is reactivated and the function applied.

5.5 Parallel Lisp system

The best way of defining a parallel Lisp system is to take a definition of a Lisp interpreter and identify where the parallelism can be derived. The main components of a Lisp interpreter are the evaluator and apply function. Below is a program in Lisp of a simple Lisp evaluator derived from Winston (51).

```
(Def Eval (S Environment)
  (Cond ((Atom S)
         (Cond ((Equal S T) T)
               ((Equal S Nil) Nil)
               ((Numberp S) S)
               (T (Value S Environment))))
        ((Equal (CAR S) 'quote) (CADR s))
        ((Equal (CAR S) 'Cond)
         (Evalcond (CDR S) Environment))
```

```
(T (Apply (Car S)
                (Mapcar '(Lambda (X)
                                (Eval X
        environment )) (Cdr S)) Environment))))
```

The EVAL function returns a value if the S expression presented to it is an atom or the quoted value if the S expression begins with a quote. EVALCOND is called if the S expression begins with a Cond. If the S expression does not belong to the above, EVAL evaluates the elements in the expression after the first from left to right. The expression with the arguments replaced by the appropriate evaluated value is passed to APPLY. The APPLY function uses the first element in the list to get the function name that will be applied to the evaluated arguments. The scanning of arguments from left to right is done by iteration using MAPCAR. In the MAPCAR expression EVAL recurses on itself. The iteration can be unfolded and a simultaneous recursive call on EVAL performed. There are two arguments to EVAL, S and ENVIRONMENT. The structure of the EVAL function is similar to the parallel matrix multiplication function described in section (4.10.2). However for EVAL, it can create an arbitrary number of subtasks. The discussion presented above shows that a Lisp machine can be incorporated into the kernel of the Bath machine.

In Lisp both program and data are constructed using a list of linked cells. The problem with linked cells

is that transferring a structure from one processor to another is not efficient. The structure has to be redrawn in a compact form within a linear block before it can be transmitted. In a large structure the number of indirections needed to redraw it can be large, thus making the process inefficient. At the receiving processor a read function is required to rebuild the list which further reduces the efficiency. In a multiprocessor where there is a global shared memory in addition to the local communication path (2) this would not matter very much. A structure is passed from one processor to another just by passing the pointer to it. A possible representation for the expression can be constructed using a linear string. Moving a string is more efficient in a multiprocessor without global shared memory. However there is also a disadvantage with string representation. An operation on a list necessitates copying part or whole of the list. For example a Cons operation on list A and B requires reserving a seperate memory space where the list A.B will be written. The copying operation in itself is time consuming and allocating an arbitrary size memory space can be very demanding on memory management.

Memory space that is no longer required must be reuseable in order to prevent memory exhaustion. In a Lisp implementation on a uniprocessor garbage collection is employed to recover used cells. If linked cells were employed for program representation the same

garbage collection scheme for uniprocessor implementation can be used. Let us look at how memory management for string representation can be done. As was already mentioned recovering an arbitrary size memory space is difficult. Allocation of memory will be easier if the memory space is allocated in fixed blocks. If the space required occupies more that one block, further blocks can be allocated and chained to the previous block.

Memory management is also concerned with the allocation of task descriptors. The data receptors for the subtasks are held in the task descriptor. As previously mentioned the number of subtasks created is not fixed. The number of data receptors required is unknown because it depends on the current subexpression being evaluated. The task descriptor can be allocated a fixed size large enough for any forseeable demand. Alternatively the size of the task descriptor varies dynamically with requirement. The choice between the two methods very much depends on whether wastage of memory is more favourable than a complex and sophisticated memory management scheme which is difficult to implement.

Concurrent evaluation of the arguments does not differentiate between fine grain and large grain parallelism. It is not justifiable to make a simple subexpression into a parallel task. For example if the subexpression is (plus 2 3), the cost of setting up a

parallel task is large when compared to the evaluation of the expression. Therefore in order to maintain a high level of efficiency, only computationally large subexpression should be made into parallel tasks. The capability to differentiate between small and large subexpression must be incorporated into the expression evaluator. Whether the differentiation between subexpression is automatic or under programmers' control depends on several factors. For the scheduling to be automatic, the evaluator must be given criteria to decide whether a subexpression is small or large. The amount of computations associated with a function depends on the function definition and the size of its arguments. However a long subexpression does not necessarily represent a large computation as in the case of finding the 'car' of a fairly long list. Programming this facility into the evaluator will introduce an extra overhead to the system. It is probable that this extra overhead is not warranted. Parallel scheduling under programmers' control can be done by annotating subexpressions. It is simple for the evaluator to recognise an annotated subexpression and this is more efficient. The annotation does not alter the structure of the language significantly but it does make the programmer aware that he or she is programming a multiprocessor. One of the ideas behind using an applicative language for multiprocessor is it makes the presence of the multiple processing elements

transparent. Therefore a program written for a sequential machine can be run on a multiprocessor without modification and the same result expected. These goodies must be weighted against efficiency and the first impression is parallel scheduling should be made under programmers' control. What was not apparent before is that the speed of the program is very dependent on how good the programmer is in selecting the parallel functions. The same program can have very different execution times with different annotations.

The discussion above described a parallel Lisp interpreter that is based on applicative order reduction. A compiler that compiles Lisp program into parallel executable codes exploits parallelism by first performing a data flow analysis on the program (34). The job of deciding whether a function should be made into a parallel task or not can be programmed into the compiler. Since the analysis is done at compile time the run time task scheduler can be made more efficient.

5.6 Conclusion

From the discussions in the preceeding paragraphs, it can be summarised that there are two main points that have to be considered in order to implement an applicative language efficiently on a multiprocessor. The first is program representation. In the author's opinion, the absence of a global memory should favour string representation. With string representation, the

speed of moving the string from one part of memory to another can be increased by using a dedicated direct memory access device that controls the memory to memory move operations. The same hardware is equally adaptable for controlling the communication through the shared memory interface.

The problem of concurrency control is more complicated. Although in the previous discussion two alternative methods were offered, it is not possible to form any opinion on which approach should be adopted. A more detailed investigation possibly by experimentation is required.

Implementing applicative languages on the Bath multiprocessor is not limited to Lisp only. The model incorporated into the multiprocessor kernel should be equally applicable to other applicative language like SASL, HOPE and Backus' functional programming system (FP). This make the potential of the Bath machine comparable to the ALICE machine. However there are fundamental differences. In the Bath machine the proposed applicative proramming model is emulated by a conventional von Neumann machine. Various points were discussed on the ways of making the machine efficient. However there is still room for improvement. Better performance could possibly be attained if the virtual machine can be supported directly by hardware emulated at microcode level. The obvious advantage of emulating the model by microcode is that the overhead is reduced

thus making computation at fine grain level more attractive. This can be seen in the dataflow machine of Gurd et al (20) which is implemented using bit slice microprocessors.

This research has been an exercise in building a multiprocessor. Although the stage of a useable system was not reached, there are a few unknowns that can be answered as a result of this research. Towards the second half of this chapter a design of a general purpose machine was proposed. Also, the foreseeable problems associated with the implementation of such a system were discussed. Perhaps this is the clearest identifiable achievement of this research which can pave the way for further development.

Despite the low budget and minute research team, it has been shown that a parallel processing system can be constructed from standard board level processes with a single board efficient communication system. This hardware has been used to investigate one software methodology and the experience of this experiment has allowed a number of suggestions to be made for an incremental improvement of the system.

REFERENCE

1. John Darlington and Mike Reeve, "ALICE A multiprocessor reduction machine for the parallel evaluation of applicative languages.," Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire (Oct. 1981).

2. Robert M. Keller, Gary Lindstrom and Suhas Patil, "A loosely-coupled applicative multi-processing system," Proc. 1979 AFIPS NCC(1979).

3. Robert M. Keller and Frank C. H. Lin, "Simulated Performance of a Reduction Based Multiprocessor," IEEE Computer July 1984.

4. F. Warren Burton and M. Ronan Sleep, "Executing Functional Programs on a virtual tree of processors," Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire (Oct. 1981).

5. Nicolas Mokhoff, "Parallelism makes strong bid for next generation computers," Computer design September 1984.

6. J. Backus, "Can Programming be liberated from the von Neumann Style? A functional style and its algebra of programs," ACM 21(8)(1978).

7. Adrian Bowyer, Philip J. Willis and John R. Woodwark, "A multiprocessor architecture for solving spatial problems," The computer journal, Vol 24,No. 4,1981.

8. E. Dagless, "A multiprocessor Cyba-M," Information processing 77, Ed. B. Gilchrist, IFIP, North Holland.

9. P.C. Treleavan and R.P. Hopkins, "A recursive (VLSI) Computer Architecture," Technical Report Series No.161,March 1981, University of Newcastle upon Tyne.

10. D. A Turner, "A new implementation technique for applicative languages," Software practice and experience vol9,31-49(1979).

11. K. P. Gostelow and R. E. Thomas, "Performance of a Simulated Dataflow Computer," IEEE transaction on computers vol c-29. no 10, october 1980

12. B. W. Wah and Y. W. E. Ma, "MANIP - A multicomputer Architecture for solving combinatorial Extremum Search Problems," IEEE Trans. on Comp. vol c-33 no. 5 May 84

13. T.J.W Clarke, P.J.S Gladstone, C.D. Maclean, A.C. Norman, "Skim-The S,K,I reduction machine," Proc. LISP-80 Conf. Stanford (aug. 1980).

14. E. Horowitz and S. Sahni, Fundamentals of computer Algorithms, Computer Science Press, Maryland, 1978.

15. I. Filotti, "Parallel General Purpose Architectures," Advanced course on VLSI architecture, University of Bristol 19-30 July 1982.

16. J. B. Dennis, "Data Flow supercomputers," IEEE Computer vol. 13, no. 11 (Nov. 1980) pp. 48-56

17. R.D. Dowsing and E.L Dagless, "Design methods for digital systems Pt.1:concurrency constructs," Computers and Digital Techniques, June 1979,vol 2,no.3

18. Texas Instrument Incorporated, Designing witl TTL Integrated Circuits, McGraw-Hill, 1975.

19. D. Aspinall, E.L. Dagless and R.D. Dowsing, "Design methods for digital systems including parallelism," Electronic circuits and systems,Jan 1977,vol 1,no2

20. I. Watson and J. Gurd, "A prototype data flow computer with token labelling," National computer conferrence 1979

21. P. C. Trealevan, "Decentralised computer architecture for VLSI," Advanced course on VLSI architecture, University of Bristol 19-30 July 1982.

22. P. J. Willis, D. Milford and J. Woodwark, "Exploiting area coherence in raster scan displays," Proc. Elec. Displays 1981, 3, pp34-46, 1981

23. K. Berkling, "Reduction Language for Reduction Machines," Proc. 2nd Int. Symp. Computer Architecture, Houston (Jan 1975).

24. C.A.R. Hoare, "Communicating Sequential Process," CACM 21(8) pp 666-677 Aug 78.

25. D. May, "occam," Sigplan notices 18(4) pp 69-79 1983.

26. Robin Milner, "Synthesis of Communicating Behaviour," Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64, pp 71-83 Springer Verlag.

27. J. B. Dennis, "First Version of a Dataflow Procedure Language," Lecture Notes in Computer Science 5, p 187 1984.

28. R. M. Burstall, "Design considerations for a functional Programming Language," Proc. of Infotech State of the Art Conference, Copenhagen, 1979.

29.C.A.R. Hoare, "Communicating Sequential Processes," On the construction of Programs. Ed. R. M. McKeag and A. M. MacNaghten Cambridge University Press, 1980, pp 229-254.

30. R. C. Singleton, "On minimal graphs with maximal even girth," J. Combinatory Theory, Vol. 1, pp 306-332, 1966.

31. C.A.R. Hoare, "Proof of a Recursive Program: Quicksort," Comp. J., 14 No.4 (1971) pp 391-395.

32. N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall 1976, p 79.

33. C.A.R. Hoare, "Quicksort," Comp. J., 5 No.1 (1962), pp 10-15.

34. J. Marti and J. Fitch, "The Bath Concurrent Lisp Machine," EUROCAL '83, Lecture Notes in Computer Science 162.

35. J.W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," Math Comp, 19, pp. 297-301, 1965.

36 N.L. Biggs and M.J. Hoare, "A trivalent graph with 58 vertices and girth 9," Communication, Discrete Mathematics 30(1980), pp 299-301, North-Holland Publishing Company.

37. W.H. Burge, Recursive Programming Techniques, Addison-wesley (1975).

38. W. B. Ackerman, "Dataflow Languages," AFIPS Conf. Proc., Vol 48, NCC, New York, June 1979, pp 1087-1095.

39. W.B. Ackerman, "Dataflow Languages," IEEE Computer 15(2), p.15 (Feb. 1982).

40. J.R.W. Glauert, "High Level dataflow Programming Distributed Computing," Apic studies in Data Processing No. 20, Ed. F.B. Chambers, D.A. Duce and G.P. Jones, pp 43-53, 1983.

41. J.R. Kennaway and M. R. Sleep, "Towards a Succeeosr to von Neumann," Apic studies in Data Processing No. 20, Ed. F.B. Chambers, D.A. Duce and G.P. Jones, pp 125-138, 1983.

42. R.H. Perrot, "Languages for Parallel Computers," On the construction of Programs. Ed. R. M. McKeag and A. M. MacNaghten Cambridge University Press, 1980, pp 255-281

43. R.E. Millstein, "Control Structures in ILLIAC IV FORTRAN," CACM, 16,10, pp. 622-627, 1973.

44. E.W. Dijkstra, "Co-operating Sequential Processes in Programming Languages," ed. F. Genuys, Academic Press, New York, pp 43-112, 1968.

45. P. Brinch Hansen, The Architecture of Concurrent Program, Prentice Hall, Inc, Englewood Cliffs, New Jersey.

46. G.H. Barnes, R.M. Kato, P.J. Kuck, D.L. Slotnick and R.O. Stokes, "The ILLIAC IV Computer," IEEE Trans. Computer, C-17, pp. 746-757, 1968.

47. S.A. Holland and C.J. Purcell, "The CDC Star-100: A large scale network oriented computer system," Proc. 1971 IEEE Conferences, pp 55-65, 1971.

48. R.M. Russell, "The CRAY-1 computer system," CACM 21, 1 pp 63-72, 1978.

49. S.F. Reddaway, "DAP- A Distributed Array Processor," Proceedings of 1st. ACM Symposium on Computer Architecture, Dec 1973.

50. W. Meschach, "Dataflow IC makes short work of tough processing chores," Electronic Design, May 17 1984, Vol. 32, No. 10.

51. P. H. Winston and B. K. P. Horn, LISP, Addison-Wesley, 1981.

52. H. T. Kung, L. M. Ruane and D. W. L. Yen, "A Two-Level Pipelined Systolic Array For Convolutions," VLSI Systems and Computations, Ed. H. T. Kung, B. Sproull and G. Steele, Coputer Science Press 1981. pp255-264.

53. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer- Designing an MIMD Shared Memory Parallel Computer," IEEE Trans. on Computers. Feb 83 Vol. C-32 NO. 2

54. Olivier Roubine and Jean-Claude Heliard, "Parallel Processing in ADA," On the construction of Programs. Ed. R. M. McKeag and A. M. MacNaghten Cambridge University Press, 1980, pp 193-212.

55. P. J. Peters, "Tree machines and divide and conquer algorithms," Lecture Notes in Computer Science 11, Conpar 81, Springer-Verlag, pp 25-36.

56. E. H. Wold and A. M. Despain, "Pipeline and Parallel Pipeline FFT Processors for VLSI implementations," IEEE Trans. on Comp. vol c-33 no 5 May 84.

57. E. Horowitz and A. Zorat, "Divide and Conquer for Parallel Processing," IEEE Trans. on Computers, June 83, C-32 No. 6 pp 582-585.

58. Motorola, MC68000 16-bit microprocessor user's manual, Prentice Hall 1982

59. Motorola Inc, MC68000 data sheets, 1981

Appendix A

MC68000 signals

The input and output signals are functionally organised into groups. Figure A.1 shows the various signals and their respective group. Basically the signals of the MC68000 are the same as other microprocessors which comprise of the address bus, data bus and the control bus. The MC68000 provides more signals in the control group compared to an eight bit microprocessor.

Address bus

The address bus is 23 bits (A1 - A23). The bus is unidirectional and can be tri-stated. The address bus supplies the address in a memory reference operation. During interrupt the address line A1,A2 and A3 signify the current interrupt level being processed. Address lines A4 to A23 are set to logic high.

Data bus

The data bus is 16 bits wide. The bus is bidirectional and can be tri-stated. The data bus can read or write in either word or byte length. Data lines D0 - D7 are used to supply a vector number in the interrupt acknowledge cycle.

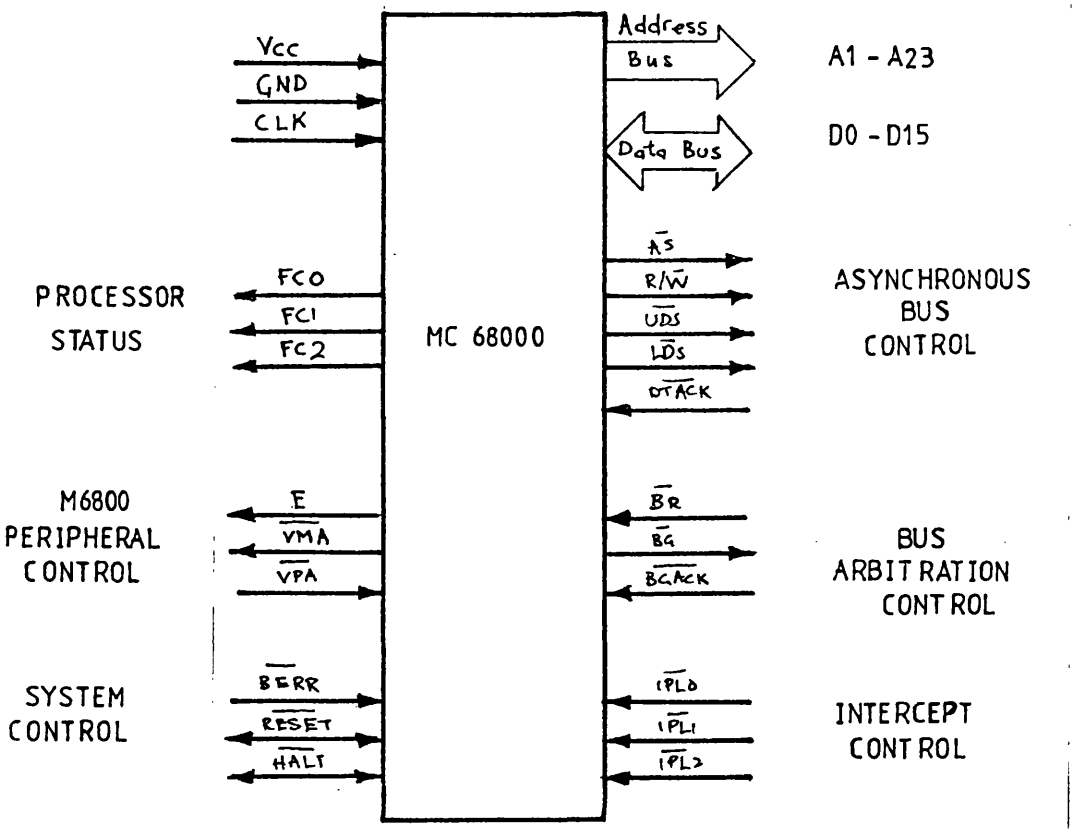


FIG. A.1 MC 68000 SIGNALS

Control bus

The control bus can be further classified into various functional subgroups. The various subgroups that form the control functions are described below:

Asynchronous bus control

The MC68000 allows asynchronous data transfers. The following signals control the asynchronous transfer: address strobe, read/write, upper and lower data strobes and data transfer acknowledge.

Address Strobe ($\overline{AS}$)

This signal is use to indicate to the memory device that there is a valid address on the address bus. This is necessary to differentiate the interrupt cycle which uses the address line A1,A2 and A3 to indicate the interrupt level.

Read/$\overline{Write}$ (R/$\overline{W}$)

The read/write signal indicates the direction of the transfer.

Upper and Lower Data strobes ($\overline{UDS}$,$\overline{LDS}$)

The MC68000 allows data transfer at word and byte levels. With a byte transfer it is necessary to specify whether the transfer is from the lower byte or the upper byte. The conditions of the UDS and LDS signals in relation to the transfer is as follows:

$\overline{UDS}$	$\overline{LDS}$	R/$\overline{W}$	
1	1	x	D0 - D15 invalid
0	0	1	D0 - D15 read
1	0	1	D0 - D7 read
0	1	1	D8 - D15 read
0	0	0	D0 - D15 written
1	0	0	D0 - D7 written
0	1	0	D8 - D15 written

Data Transfer Acknowledge ($\overline{DTACK}$)

The $\overline{DTACK}$ signal is an input. The assertion of $\overline{DTACK}$ signals the processor that the data transfer is completed. During a read cycle $\overline{DTACK}$ causes the data to be read and the bus cycle to terminate. $\overline{DTACK}$ also causes the write cycle to terminate.

Bus Arbitration Control

There are three signals that make up this group. They are Bus Request($\overline{BR}$), Bus Grant($\overline{BG}$) and Bus Grant Acknowledge($\overline{BGACK}$). The functions of these signals is to coordinate the release of bus control by the processor to device that can be the bus master.

Interrupt Control ($\overline{IPL0}$, $\overline{IPL1}$, $\overline{IPL2}$)

These are encoded inputs for identifying the priority levels of the interrupting device.

System Control

There are three input lines that form the system control. The Bus Error($\overline{\text{BERR}}$) input is use to signal an error condition to the processor. The error condition could be raised by the following condition:

1. nonresponding device
2. failure to acquire interrupt vector
3. illegal access request as determined by a memory management unit.

The reset($\overline{\text{RESET}}$) input is a bidirectional signal line. The application of the reset signal externally causes the processor to reset its internal state. The execution of a reset instruction internally generates the reset signal which can be used to reset external device.

The Halt($\overline{\text{HALT}}$) signal is also a bidirectional line. The assertion of this signal externally will cause the processor to stop at the completion of the current bus cycle. The halt signal is generated internally when the processor stopped due to a double bus fault. In the halted state, all the control signals are inactive and all tri-state lines in the high impedance state.

M6800 Peripheral Control

These control signals enables the MC68000 to be used with synchronous M6800 peripheral devices.

Enable(E)

The enable signal is common to all M6800 peripheral devices.

Valid Peripheral Address($\overline{VPA}$)

This input is used to indicate to the processor that the current memory of device addressed should be treated as M6800 peripherals. This input is also used to generate automatic vectoring.

Valid Memory Address($\overline{VMA}$)

This is an output and is used to indicate to the M6800 peripheral that there is a valid address and the processor is synchronised to the enable signal.

Processor Status (FC0,FC1,FC2)

These are output lines and are used to indicate the processor state. The function code outputs are only valid when address strobe is true. The various processor states are as follows:

FC2	FC1	FC0	Cycle Type
0	0	0	-
0	0	1	User data
0	1	0	User program
0	1	1	-
1	0	0	-
1	0	1	Supervisor data
1	1	0	Supervisor program
1	1	1	Interrupt acknowledge

- indicates undefined or reserved.

Clock(CLK)

The clock input is TTL compatible.

MC68000 bus operations

All bus operations are synchronised to the processor clock states. The clock is divided internally to generate eight states S0 to S7. The timings of the signals are linked to these states.

Read cycle

The processor receives data from memory or peripheral during a read cycle. A read instruction can specify the size of data to be byte, word or long word. The condition on the lower and upper data strobe signals indicate to the memory or peripheral the size of the transfer. If the instruction specifies a byte operation either data strobe but not both is asserted. This determine whether the upper or lower byte is to be read. If the instruction specifies a word or long word operation both data strobes are asserted thus reading both bytes simultaneously. In long word transfer two successive memory read operations are done.

The sequence of actions involved in a memory read operation is now described . The processor which acts as the bus master generates the following signals:

1. Set R/$\overline{W}$ to read;
2. Place function code on FC0-FC2;
3. Place address on A1-A3;
4. Assert address strobe;
5. Assert upper and lower data strobes accordingly.

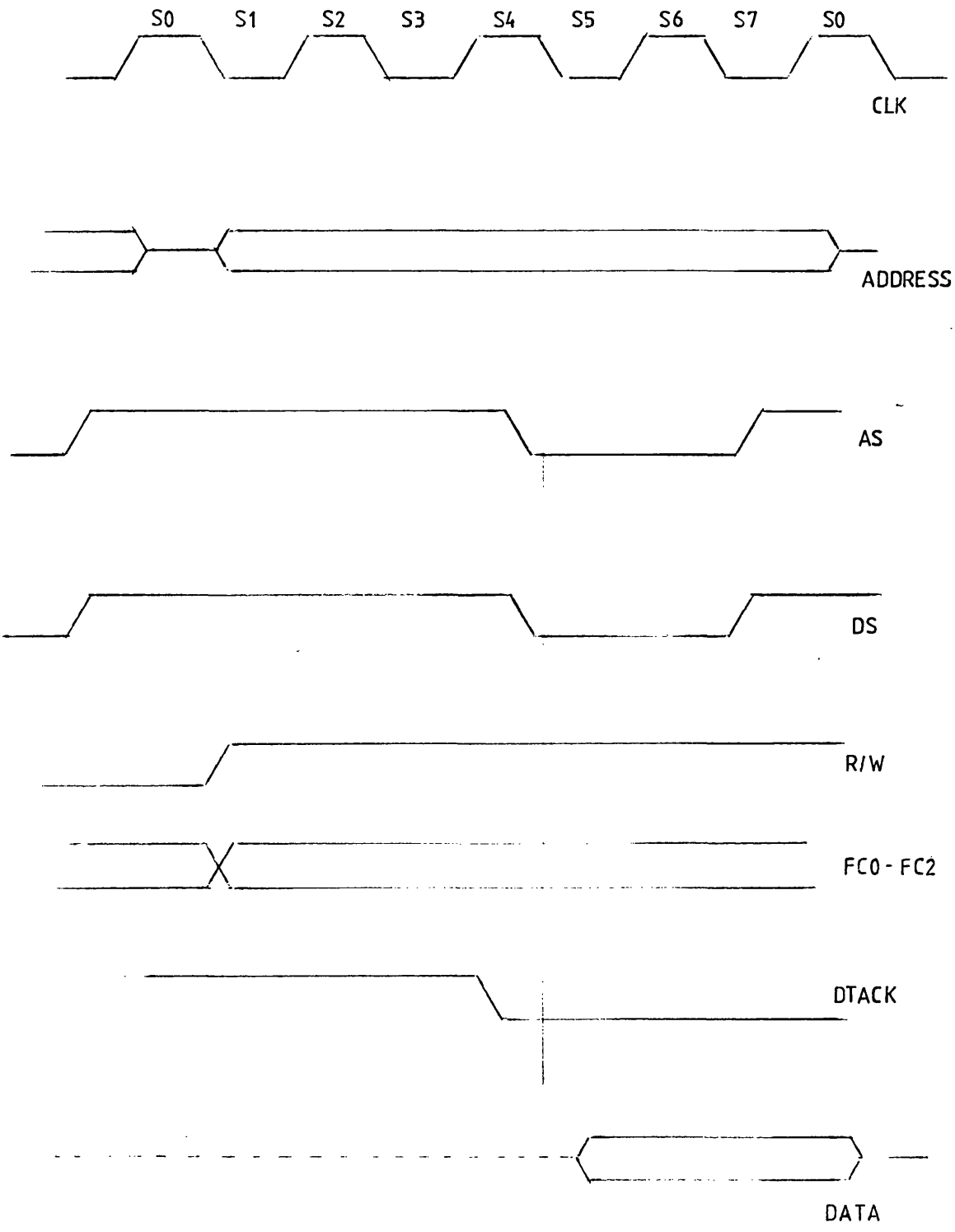


FIG. A.2 PROCESSOR READ TIMING

The memory or peripheral actions are as follows:

1. Decode address;
2. Place data on data bus;
3. Assert data transfer acknowledge($\overline{DTACK}$).

On recognising the assertion of $\overline{DTACK}$ the processor initiates to acquire the data. The data is latched and the data and address strobes negated. The negation of the data and address strobes signals the memory or peripheral to terminate the cycle. In state S0 the address bus is in a high impedance state. Lines FC0-FC2 generate the appropriate code according to the address space that is going to be accessed. The R/$\overline{W}$ line is set high indicating a read operation. In state S1 the address bus outputs a valid address.

In state S2, the address strobe($\overline{AS}$) and the appropriate data strobes are asserted. The memory or peripheral device is selected in this state. The device places data on the data bus and at the same time assert $\overline{DTACK}$. If $\overline{DTACK}$ is not asserted before the set up time at the end of state S4, the wait state is subsituted for states S5 and S6.

The address and data strobes are negated at the end of state S7. The memory or peripheral device is deselected. The address bus, R/$\overline{W}$ and function code lines are held valid through state S7 to ensure proper operation.

Write cycle

The sequence of actions for a write cycle is similar in some respect to that of the read cycle. However there are a few disimilarities as described below. The R/$\overline{W}$ line is set low to indicate a write operation. The R/$\overline{W}$ line is pulled low in state S2 and will remain in this state through to the end of state S7. The data strobes are asserted in state S4. The data that is to be written to memory is placed on the data bus one state earlier in state S3.

Read Modify Write Cycle

In read modify write cycle, a byte read operation is followed by a write operation. The difference between this cycle and a normal read and write cycle is that the bus is not released after the read operation. This is done by the processor holding the address strobe asserted.

Interrupt processing

The MC68000 can be in either of the following states: normal, exception or halted. Interrupts, trap instructions and other exceptional conditions can cause the MC68000 to go into exception state. The MC68000 provides seven levels of interrupt priorities. An unlimited number of interrupt sources can be serviced within an interrupt priority level. The interrupt priority levels are numbered from one to seven. Level seven is the highest priority. The status register

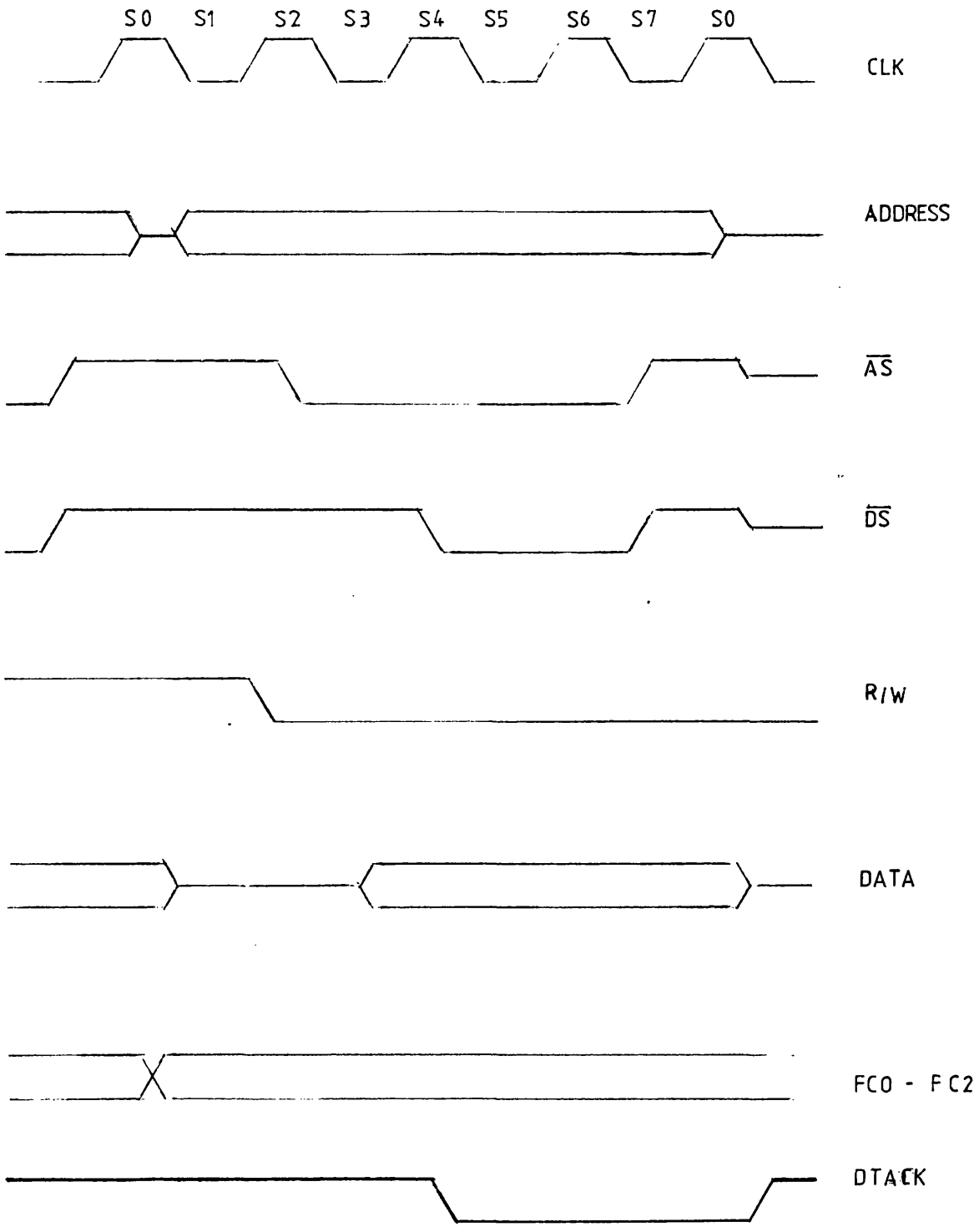


FIG. A.3 PROCESSOR WRITE TIMING

contains a three bit mask which indicates the current priority level. Only interrupts of priority level higher than the current level are serviced. An interrupt is made by encoding the required interrupt priority level on the interrupt lines. On arrival of an interrupt request, the interrupt is not serviced immediately but made pending. The interrupt is detected in between instruction execution. The interrupt request is ignored if the requested interrupt has the same or lower priority than the present processor state. A pending interrupt request which has a higher priority level will start an exception processing sequence. The processor responds by saving the status register on the stack, setting the processor state to supervisor, setting the trace mode to off and updating the interrupt priority level to the interrupt level being serviced. The processor acknowledges the interrupt to the external device by sending out an interrupt acknowledge code on the FC0-FC2 lines and the interrupt level being processed on A1,A2,A3 lines. The external device must respond by asserting $\overline{\text{DTACK}}$ or $\overline{\text{VPA}}$. If $\overline{\text{DTACK}}$ is asserted, the external device must also supply the interrupt vector on the data bus. If $\overline{\text{VPA}}$ is asserted, the vector is generated internally by the processor. If the bus error line is asserted, the processor will assume that a spurious interrupt has occur. The processor jumps to the location defined by the spurious interrupt vector for error processing.